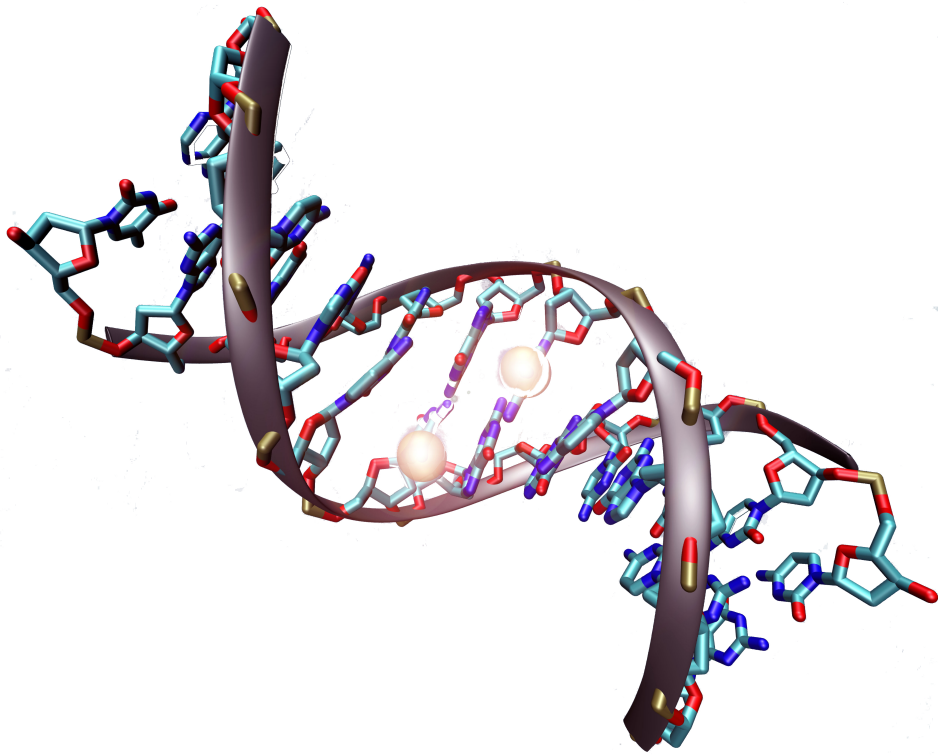
Ioana Bica

Comparative Analysis of Neural Network Architectures for Epigenetics Inference

Part II Computer Science Tripos

Churchill College, 2017

May 19, 2017



The cover page image illustrates the epigenetic mechanism of DNA methylation, which involves the addition of a methyl group CH_3 to a guanine or cytosine nucleotide. DNA methylation has a crucial role in the development of organisms. Source: Max Plank Institute for Informatics.

Proforma

Name:	**Ioana Bica**
College:	**Churchill College**
Project Title:	**Comparative analysis of neural network architectures for epigenetics inference**
Examination:	**Computer Science Tripos – Part II (June 2017)**
Word Count:	11984[1]
Project Originator:	Petar Veličković
Supervisors:	Dr Pietro Lio' and Petar Veličković

Original aims of the project

The aims of the project were to design and implement three different types of neural network architectures to perform inference on epigenetic data. The models were chosen such that each one of them explores different properties of the epigenetic data in order to overcome the problem of having sparse and imbalanced datasets. Clustering algorithms would also be implemented as a form of data pre-processing. All of these models were subsequently compared in order to assess their abilities on pattern recognition tasks.

Work completed

The success criteria for the project have all been achieved and several extensions have been implemented. The models implemented are a multilayer perceptron optimised for data sparsity, a recurrent neural network that takes advantage of the sequential form of the epigenetic data and a superlayered neural network that explores the heterogeneity of this data. Hierarchical clustering and k-means clustering algorithms were also developed in order to determine the input to the superlayered neural network. The use of neural networks to study epigenetics has been successful and the results obtained can benefit medical research in this field.

[1]This word count was computed using TEXcount, `http://app.uio.no/ifi/texcount/`.

Special difficulties

None.

Declaration

I, Ioana Bica of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

_____

Signed

_____

Date

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisors **Dr Pietro Lio'** and **Petar Veličković** for their invaluable advice and help throughout the course of the project and for their useful feedback on this dissertation. Additional thanks go to **Hui Xiao** for helping me understand epigenetics and for providing me with the datasets.

I am also grateful for the guidance and help offered throughout my studies by **Dr John Fawcett.**

Moreover, I would like to thank my **family** for their unwavering support and understanding. Finally, I am grateful to my dear friend, **Malavika Nair**, for countlessly proofreading this dissertation and for offering me moral support and encouragement throughout the entire year.

Chapter 1

Introduction

The dawn of epigenetics has offered the field of molecular biology ways to explore previously inexplicable changes in the physical structure of the DNA. The increasing availability of epigenetic datasets, however, has concurrently presented the challenge of extracting features from this information-dense data efficiently and accurately using computational tools.

This dissertation describes the implementation, comparative analysis and evaluation of three different neural network (NN) architectures. Each of the three NNs—multilayer perceptron (MLP), recurrent neural network (RNN) and superlayered neural network (SNN)—performs pattern recognition on epigenetic data by exploiting different properties of this data.

1.1 Motivation

Epigenetics explores how modifications in organisms arise from variations in gene expression[1]. Amongst the epigenetic mechanisms that affect these changes in the phenotype of cells are DNA methylation and histone modifications. As seen in Figure 1.1, the interactions between the epigenetic factors, epigenetic mechanisms and the way they affect the gene expression levels are complex and often difficult to interpret.

Epigenetic data were chosen for this project due to the vital role they play in the cause and diagnosis of diseases, as well as in the development of organisms. Since the preliminary signs of a disease will be noticeable first in the epigenetic mechanisms,[7,14,26] understanding epigenetic data better and being able to make useful predictions from them will be paramount for medical research and subsequent diagnosis.

The move from data produced by genome projects to final diagnostic interpretation in medical research, however, requires a powerful machine learning tool capable of supervised learning.

The incorporation of NNs for supervised learning tasks in particular has skyrocketed in both use and performance. Due to the development of new architectures and

[1] Gene expression is the process of synthesising proteins based on genetic information.

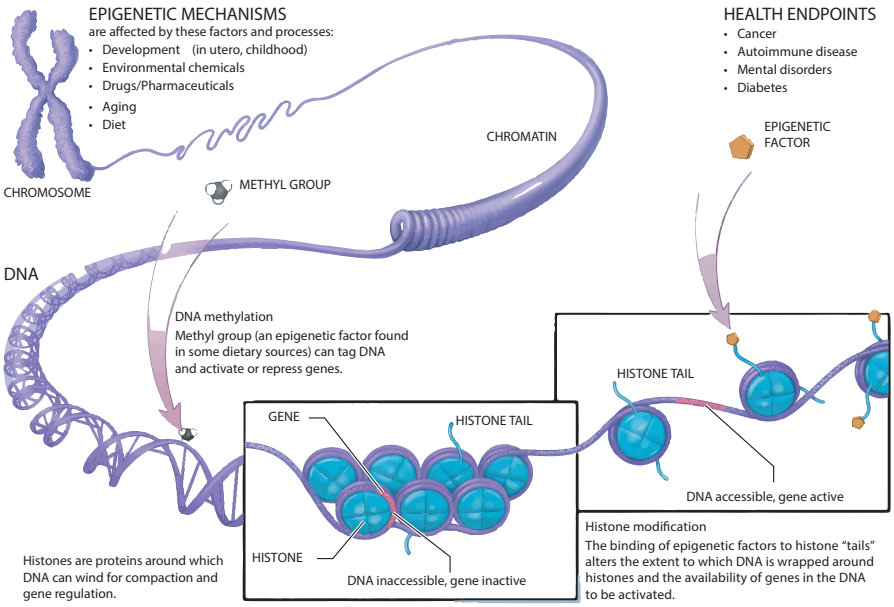


Figure 1.1: Illustration of two epigenetic mechanisms—DNA methylation and histone modifications—and their effects on the physical structure of the DNA. Source: National Institutes of Health

the availability of large quantities of training data, deep learning models are now capable of achieving state-of-the-art performance on increasingly complex problems such as image recognition,[16] speech processing,[11] and machine translation.[30]

The main reason for choosing NNs to analyse epigenetic data is their exceptional ability of handling data with non-linear relationships. By exploring the high dimensionality of the epigenetic data, NNs could achieve exceptional results on pattern recognition tasks.

Additionally, by operating directly on the raw input, such models remove the need for explicit feature engineering. Once the initial NN architecture has been developed for one type of epigenetic data, it can be applied to any epigenetic dataset, provided it has the same input structure.

These advantages provide sufficient evidence towards choosing NNs to perform inference on epigenetic data. Nevertheless, this represents a very challenging endeavour, as outlined in the next section.

1.2 Challenges

The greatest challenges in handling epigenetic data are related to the sparsity of the datasets. Due to the high cost of performing experiments to obtain epigenetic data, the available datasets contain a limited number of training examples. As a consequence, there is a high risk of overfitting in the machine learning models.

Moreover, epigenetic datasets are unbalanced: certain classes are overly represented compared to others. Therefore, it is essential to use appropriate performance metrics to account for the imbalance in the testing data.

Epigenetic data are also information dense: there can be thousands of input features of interest per training example. Considering that it is practically infeasible to incorporate all these input features in a single model, additional pre-processing of the data is required. However, the property of having a large number of input features can be used to overcome the problem of having small datasets.

Additionally, choosing the most suitable models to analyse epigenetic data can be problematic. With the development of so many different NN architectures, it is critical to understand their properties to make the most task-appropriate selection.

1.3 Related work

This section will discuss the basic principles behind existing deep learning models used on epigenetic data and the ways these models are used on more generic datasets.

1.3.1 Deep learning on epigenetic datasets

Multilayer perceptrons have been applied to identify cancer biomarkers from DNA methylation [31] and to predict DNA methylation-based forensics. [34]

Moreover, Angermueller *et al.* [2] proposed an architecture that combines an RNN with a convolution neural network (CNN) to predict DNA methylation states in all cells, thus enabling the study of sources of epigenetic diversity.

The analysis of histone modifications has also provided valuable results. Singh *et al.* [27] designed DeepChrome, a CNN based model, to predict gene expression from interactions in histone modifications. DeepChrome has enhanced research in developing 'epigenetic drugs' for various diseases.

1.3.2 Sequence analysis using recurrent neural networks

RNNs are a special type of NN where connections between the units form cycles. This property makes them appropriate for studying sequences. In bioinformatics, RNNs have already been successfully used to predict protein structure, [4, 28] and also to identify patterns in gene expression regulation. [19, 20, 24]

1.3.3 Multilayer networks

The concept of multilayer networks arose from the need to integrate multiple types of information to understand complex systems. Multilayer networks [15] have been successfully employed to study systems from social networks [6] to epidemics. [9]

A deep learning model inspired by multilayer networks was used by Veličković *et al.* to improve the performance of NNs on sparse image datasets.[33] The proposed architecture consists of several CNN superlayers designed to study different partitions of the input data. While this approach was designed to aid clinical studies, the same idea can be applied to epigenetic data.

Chapter 2

Preparation

This chapter explores the underlying theory of the NN models implemented as well as the requirements analysis and software engineering principles applied for a successful implementation.

2.1 Introduction to neural networks

This section describes the building blocks of the NN architectures used in the project as well as the steps involved in training them. Each architecture was chosen due to its applicability to the relevant epigenetic datasets.

NNs are supervised learning models, and for the purpose of my project, they will be used solely on classification tasks.

2.1.1 Supervised learning using neural networks

Supervised learning models require a labelled training dataset, where each input feature vector $\mathbf{x} \in \mathbb{R}^m$ is associated with a label y. In classification problems, y represents one of the possible output classes $\mathcal{C} = \{C_1, ..., C_k\}$.

Given a training sequence

$$\mathbf{s} = [(\mathbf{x}_1, y_1)\ (\mathbf{x}_2, y_2)\ ...\ (\mathbf{x}_n, y_n)] \tag{2.1}$$

where each $(\mathbf{x}_i, y_i)$ is a training example, the NN learns a parametrised function $h : \mathbb{R}^m \to \mathcal{C}$, called the hypothesis, that maps the input vector $\mathbf{x}$ to the output class y.

An NN defines some function $f(\mathbf{x}; \mathbf{w}, \mathbf{b})$, dependent on the architecture, where $\mathbf{x}$ is the input vector, and $\mathbf{w}$ and $\mathbf{b}$ are the weights and biases in the network respectively. The hypothesis is then set to be $h(\mathbf{x}) = f(\mathbf{x}; \mathbf{w}, \mathbf{b})$ for some appropriately chosen $\mathbf{w}$ and $\mathbf{b}$. An optimisation algorithm is used to find $\mathbf{w}$ and $\mathbf{b}$ such that a loss function is minimised. The loss function is usually defined to measure some notion of the network's error when estimating y from $\mathbf{x}$ (given $\mathbf{w}$ and $\mathbf{b}$). This procedure is commonly referred to as "training".

After the model has learnt the function h from the training dataset, it can be used to classify previously unseen data, $\mathbf{x}'$.

2.1.2 Artificial neuron

The fundamental unit of an NN is the artificial neuron, illustrated in Figure 2.1.

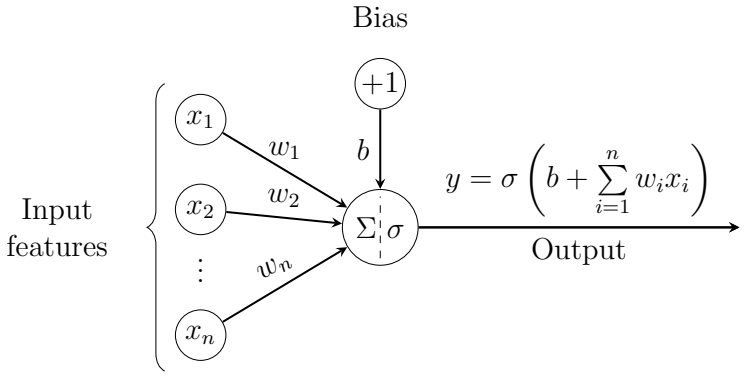


Figure 2.1: Structure of an artificial neuron.

An artificial neuron computes a linear combination of its input features and then applies an activation function $\sigma : \mathbb{R} \to \mathbb{R}$ to it:

$$y = \sigma\Big(b + \sum_{i=1}^{n} x_i w_i\Big) \tag{2.2}$$

Each input x_i has an associated weight w_i and the neuron has its own bias, b, which is used to shift the activation function on the horizontal axis.

Some common activation functions include the:

- *logistic function*

$$\sigma(x) = \frac{1}{1 + \exp(-x)}, \tag{2.3}$$

- *hyperbolic tangent* (tanh)

$$\sigma(x) = \tanh(x), \tag{2.4}$$

- *rectified linear unit* (ReLU)

$$\sigma(x) = \max(0, x). \tag{2.5}$$

Figure 2.2 illustrates the form of these activation functions and their derivatives. Generally, steeper gradients such as that of the ReLU function [23] will cause the training algorithm to converge more quickly.

The choice of the activation function also depends on the type of function the model needs to learn. For example, the logistic function is used to represent binary (Bernoulli) probabilities, while tanh is a general purpose symmetric function.

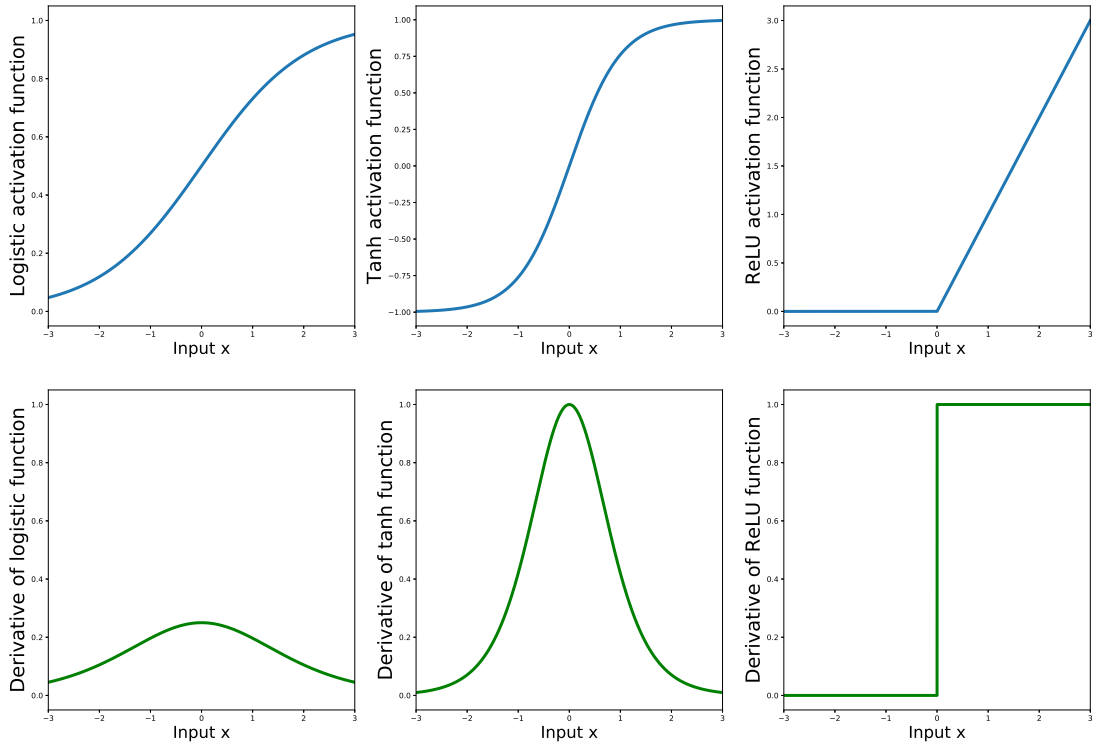


Figure 2.2: Common activation functions (top row) and their derivatives (bottom row).

An NN architecture consists of many interconnected artificial neurons, which in combination are capable of learning complex functions.

2.1.3 Multilayer perceptron

The multilayer perceptron (MLP) is a type of feedforward NN and is used to study all of the input features in the epigenetic data at once.

In an MLP, the artificial neurons are arranged in layers, as illustrated in Figure 2.3. The input layer consists of input features, while the neurons in the output layer represent predictions for the possible classes. Each neuron in a hidden layer analyses the outputs from the previous layer and computes the input to the neurons in the next layer.

Suppose that the network should learn to classify input $\mathbf{x}$ as part of class C_t, the activation for neuron y'_t in the output layer should indicate this classification. A popular choice for performing this task is the softmax activation function, which monotonically transforms any real-valued vector into a categorical probability distribution:

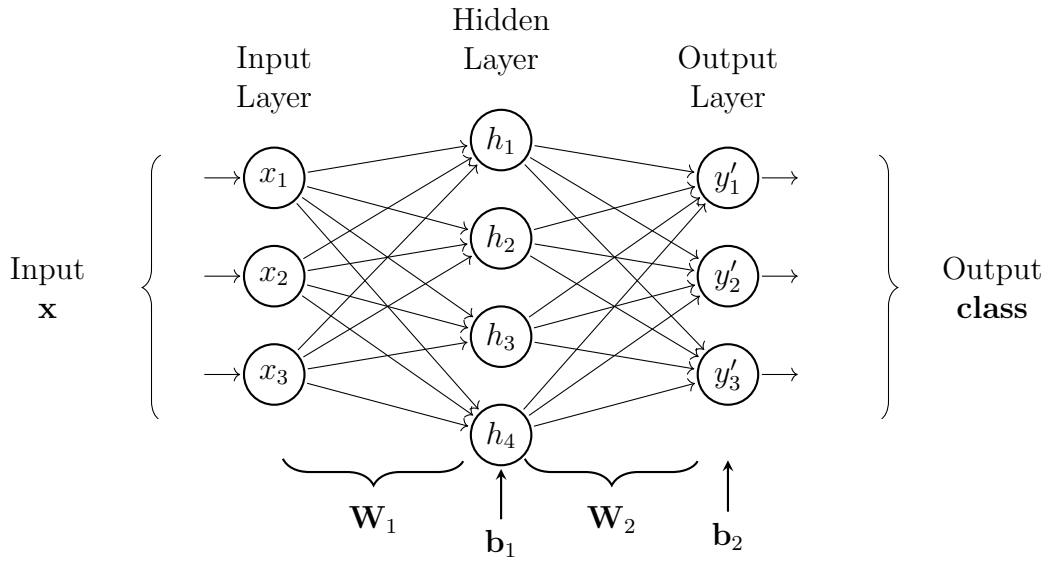


Figure 2.3: Structure and data flow of a multilayer perceptron.

$$\textbf{softmax}\,(\mathbf{y}')_i = \frac{\exp(y'_i)}{\sum_{j=1}^{k}\exp(y'_j)} \tag{2.6}$$

One-hot encoding is used to represent the ground truth probability distribution of an input example. The one-hot encoding of the output for an input vector in class C_t has the form:

$$\mathbf{y} = [y_1\ y_2\ \ldots\ y_k] \tag{2.7}$$

where

$$y_i = \begin{cases} 1 & \text{if } i = t \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

2.1.4 Neural network operations on tensors

This section describes the operations of an NN on the input data as operations on tensors to maintain consistency with TensorFlow, the library I used to implement the architectures.

Consider the MLP architecture in Figure 2.3, and assume that it uses the ReLU activation function. For an input vector $\mathbf{x}$, the output $\mathbf{y}'$ can be computed by applying the operations described in Figure 2.4.

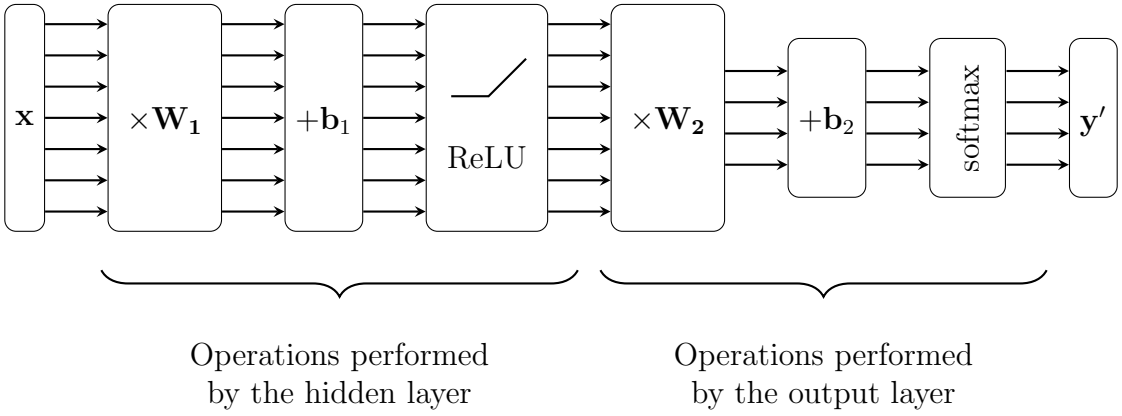


Figure 2.4: Representation of the operations performed in an NN as operations on tensors.

In mathematical notation, this is equivalent to:

$$\mathbf{y}' = \mathbf{softmax}(\mathbf{W}_2\mathrm{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) \tag{2.9}$$

For N_L neurons in layer L, the weights for the input to layer n and biases for layer n can be represented as tensors of shapes (N_{n-1}, N_n) and (1, N_n) respectively.

By using this representation, the model can be employed to process multiple training examples at once. For E examples, I input features and O output classes, this is achieved by representing the input and output of the network as tensors of shapes (E, I) and (E, O) respectively.

This is important for training, since it will make it possible to use many training examples at once for optimising the weights and biases.

2.1.5 Training a neural network

The first step in training an NN is to **initialise the weights and biases**. Initialisation schemes will be discussed in more detail in the Implementation chapter since they are specific to each architecture.

The algorithm used for optimising the weights and biases is **mini-batch gradient descent**, which performs several training iterations until the parameters converge. At each iteration, the weights and biases are updated based on the gradient computed over mini-batches of the training data. This represents a 'golden middle' between updating over the entire training set (which can be slow and memory-intensive) and updating over one training example at a time (which can lead to unstable convergence).

At iteration t, a mini-batch $\mathcal{B} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_m, \mathbf{y}_m)\}$ is selected, where each example $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{B}$ consists of the input features $\mathbf{x}_i$ and the one-hot encoding of the corresponding class $\mathbf{y}_i$.

The input features $\mathbf{x}_i$ are **forward propagated** through the network to compute the activations in the output layer. The output $\mathbf{y}'_i$ is then obtained by applying the softmax function to these activations.

The **cross-entropy loss function**[1] is used to compute the difference between the true distribution of the output $\mathbf{y}_i$ and the computed output $\mathbf{y}'_\mathbf{i}$ and is defined as:

$$\mathcal{L}(\mathbf{w}, \mathbf{b}, \mathbf{x}_i, \mathbf{y}_i) = -\sum_{j=1}^{k} y_j \log(y'_j) \tag{2.10}$$

where y_j and y'_j are the components of $\mathbf{y}_i$ and $\mathbf{y}'_i$ respectively. The cross-entropy loss function is differentiable in the weights and biases, thus allowing for gradient descent optimisation to be applied.

The weights and biases are updated using mini-batch gradient descent as follows:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{B}} \frac{\partial \mathcal{L}}{\partial \mathbf{w}}\bigg|_{\mathbf{w}_t} \tag{2.11}$$

$$\mathbf{b}_{t+1} \leftarrow \mathbf{b}_t - \alpha \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{B}} \frac{\partial \mathcal{L}}{\partial \mathbf{b}}\bigg|_{\mathbf{b}_t} \tag{2.12}$$

where $\mathbf{w}_t$ and $\mathbf{b}_t$ are the weight and bias vectors respectively at iteration t and $\mathcal{L} = \mathcal{L}(\mathbf{w}, \mathbf{b}, \mathbf{x}_i, \mathbf{y}_i)$.

The learning rate α is a hyperparameter[2] which decides the amount by which the direction of the gradient is followed at each iteration. Appropriate techniques are needed for choosing this hyperparameter correctly, since its value is critical in the convergence of gradient descent.

The gradient of the loss function with respect to the weights and biases is obtained using **backpropagation**. Starting from the output layer, where the gradient can be directly computed from the loss function, the gradient at each hidden layer is obtained using the chain rule on the gradient of its succeeding layer. The process is repeated until the input layer is reached.

2.1.6 Making predictions using neural networks

After training the NN, the model can be used to make predictions on previously unseen examples.

Given the output $\mathbf{y}'$ as a probability distribution over classes for the input $\mathbf{x}$, the NN predicts the class with the highest probability, using the formula:

$$C = \underset{C_i \in \{C_1, C_2, \ldots, C_k\}}{\arg\max} \mathbb{P}(\mathbf{x} \text{ in class } C_i) \tag{2.13}$$

[1] The use of the cross-entropy as a loss function maximises the probability of the correct class because $y_i = 0$ for i such that $\mathbf{x} \notin C_i$

[2] A hyperparameter is a parameter associated with the training process.

2.1.7 Recurrent neural networks

Instead of analysing all the input elements at once like the MLP, an RNN processes the input features as a sequence of timesteps as shown in Figure 2.5. This model can be used to take advantage of the sequential structure of the epigenetic data.

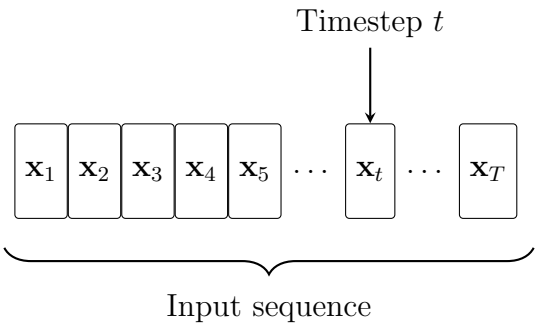


Figure 2.5: Processing of the input feature as a sequence of timesteps in an RNN

Compared to the MLP where the data flows in a single direction, the RNN architecture allows for cycles, as illustrated in Figure 2.6.

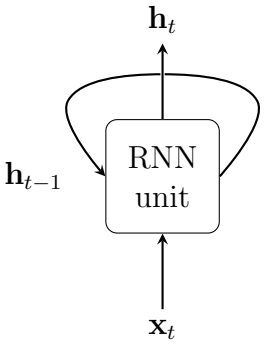


Figure 2.6: The data flow in a simple RNN.

The RNN unit is a part of the RNN and consists of interconnected neurons that extract features from the input. At each timestep t, the RNN unit computes $\mathbf{h}_t$ based on the current input $\mathbf{x}_t$ and the previous output $\mathbf{h}_{t-1}$. The unrolled computation of the RNN is illustrated in Figure 2.7.

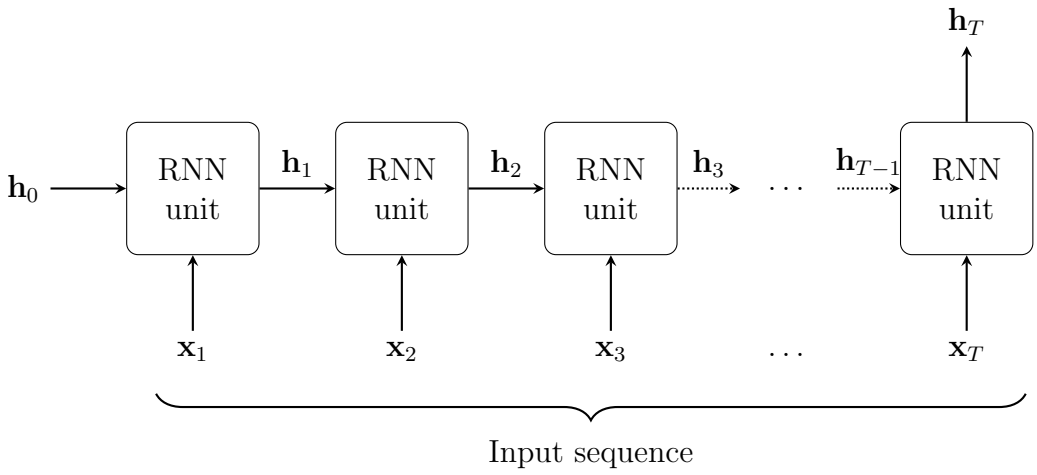


Figure 2.7: Unrolled computational graph of the RNN

An important characteristic of the RNN is that the parameters within an RNN unit are shared, so that the same weights and biases are used to perform the computation at each timestep. Hence, the number of parameters the RNN needs to learn is considerably smaller in comparison to those in an MLP. This represents an advantage for epigenetic datasets that do not have enough training examples to properly optimise a large number of parameters.

2.1.7.1 Long Short-Term Memory Units

The long computational paths in the RNN architecture cause the problem of vanishing gradients: the gradients of some weights may become too small, and updates to these parameters do not produce any noticeable difference in the output. Further detail on how vanishing gradients arise is provided in Appendix A.

The Long Short-Term Memory (LSTM) Unit architecture was developed to solve the problems posed by vanishing gradients.[12] The LSTM Unit introduces the memory cell, which maintains features between the timesteps. The features learnt depend on the previous output, the current input as well as the operations of some input, forget and output gates.

The memory cell solves the problem of vanishing gradients by allowing the gradients to propagate unmodified through the network and by controlling the gradient values that are backpropagated through the gates. The operations of the LSTM unit are described in Figure 2.8.

The LSTM computes a new set of features based on the current input $\mathbf{x}_t$ and previous output $\mathbf{h}_{t-1}$ as follows:

$$\mathbf{ft}_t = \tanh(\mathbf{W}_{ft}\mathbf{x}_t + \mathbf{U}_{ft}\mathbf{h}_{t-1} + \mathbf{b}_{ft}) \tag{2.14}$$

where $\mathbf{W}_{ft}$, $\mathbf{U}_{ft}$ are the weights associated with $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ respectively and $\mathbf{b}_{ft}$ is the bias of the gate. The tanh activation function provides a symmetric, zero-centred response.

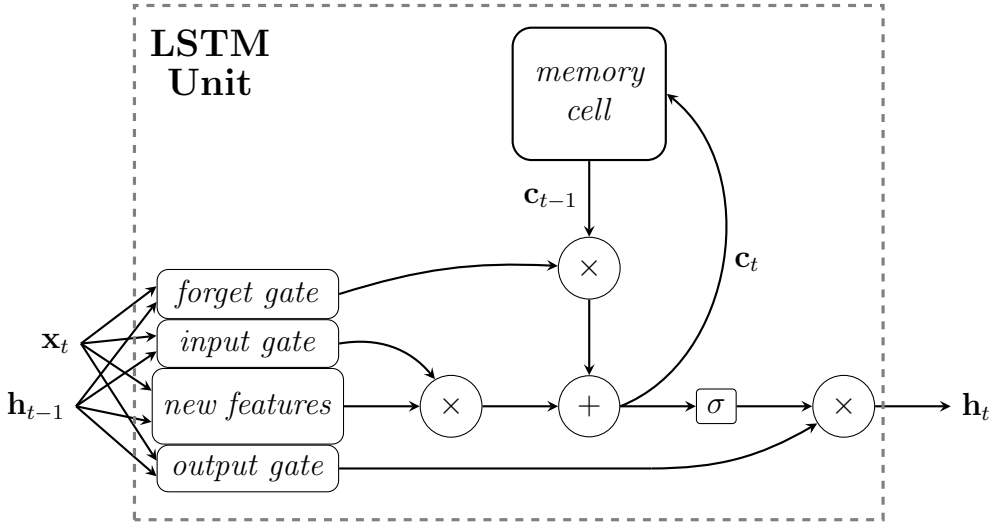


Figure 2.8: Operations performed by the LSTM unit on the input features $\mathbf{x}_t$ and hidden state $\mathbf{h}_{t-1}$ at timestep t.

The input, forget and output gates decide which of the new features are used to update the memory cell and to output at the current time step. The logistic function is a natural choice for a differentiable gating mechanism, given its output range.

The gates in the LSTM unit modulate the interactions between $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ as follows:

- the input gate decides how to modify the state of the memory cell based on the new data and previous output:

$$\mathbf{i}_t = \text{logistic}(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i) \tag{2.15}$$

- the forget gate computes which features to remove from the memory cell:

$$\mathbf{f}_t = \text{logistic}(\mathbf{W}_f\mathbf{x}_t + \mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{b}_f) \tag{2.16}$$

- the output gate allows current information to affect future timesteps:

$$\mathbf{o}_t = \text{logistic}(\mathbf{W}_o\mathbf{x}_t + \mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{b}_o) \tag{2.17}$$

where $\mathbf{W}_i$, $\mathbf{W}_f$,$\mathbf{W}_o$, $\mathbf{U}_i$, $\mathbf{U}_f$, $\mathbf{U}_o$ are the weights and $\mathbf{b}_i$, $\mathbf{b}_f$, $\mathbf{b}_o$ are the biases of the gates.

At each timestep, the state of the memory cell is updated to forget features that are no longer relevant and to incorporate information from the newly computed features:

$$\mathbf{c}_t = \mathbf{ft}_t \otimes \mathbf{i}_t + \mathbf{c}_{t-1} \otimes \mathbf{f}_t \tag{2.18}$$

where $\otimes$ is the element-wise product.

Moreover, the output of the LSTM unit is computed such that both the memory cell and current state are allowed to influence future timesteps:

$$\mathbf{h}_t = \tanh(\mathbf{c}_t) \otimes \mathbf{o}_t \tag{2.19}$$

By controlling the information that is retained and passed further on at each timestep, the LSTM unit is capable of learning long-term dependencies in the data.

2.1.8 Superlayered neural network

The superlayered neural network (SNN) is a **novel architecture** inspired by multilayer network theory that consists of two multilayer perceptrons (each forming a superlayer) with cross-connections added between them, as illustrated in Figure 2.9.

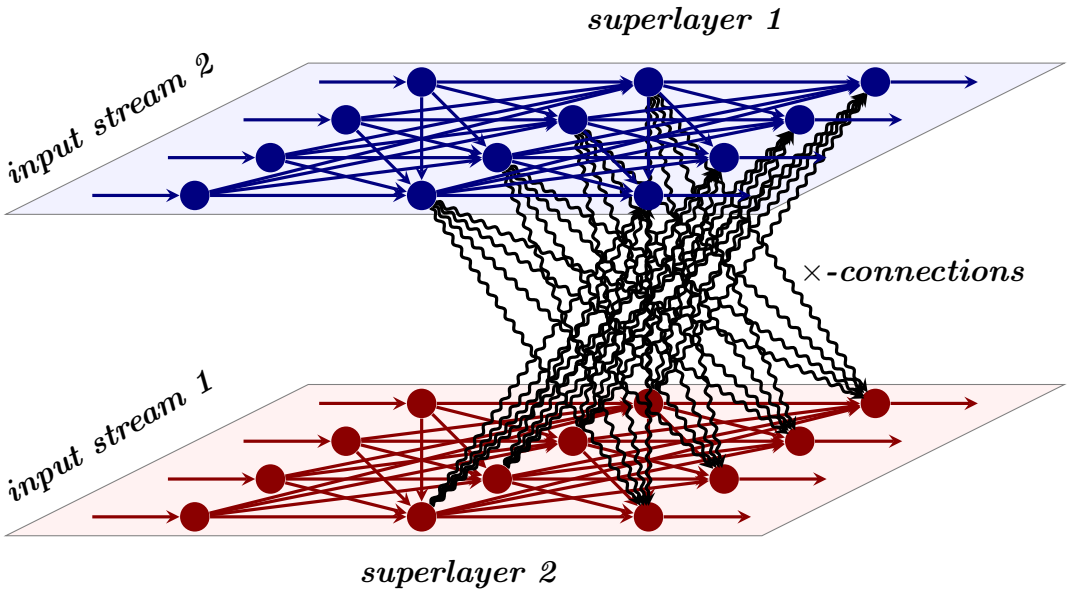


Figure 2.9: The architecture of an SNN with two superlayers.

The SNN allows for the heterogeneity of epigenetic data to be exploited. Each superlayer receives as input a partition of features sharing similar characteristics. These partitions can represent different types of input data or clusters within the same type of input data. Cross-connections in the architecture provide a way of exploring the interactions between these input partitions.

A training example for this type of network has the form:

$$\left([\mathbf{x}^{(1)}, \mathbf{x}^{(2)}], \mathbf{y}\right) \tag{2.20}$$

where $\mathbf{x}^{(1)}$ is the input to the first superlayer and $\mathbf{x}^{(2)}$ is the input to the second superlayer.

2.2 Introduction to clustering

Clustering is used as a pre-processing step to exploit the underlying structure of the epigenetic data and to select the inputs for the SNN. Clustering is an unsupervised learning task that learns the structure of the dataset without using explicit labels.

Given a dataset $\mathcal{D}$ with n input feature vectors $\mathcal{D} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, where $\mathbf{x}_i \in \mathbb{R}^m$, a clustering algorithm finds a partition of $\mathcal{D}$ into k clusters such that feature vectors in the same cluster are more similar to each other than to feature vectors in different clusters (under some notion of similarity).

In this project, clustering is used to identify co-expressed genes. For this purpose, the Pearson correlation coefficient (PCC) represents an appropriate measure of similarity. Given feature vectors $\mathbf{x} = [x_1\ x_2\ \dots\ x_m]$ and $\mathbf{y} = [y_1\ y_2\ \dots\ y_m]$, the PCC between $\mathbf{x}$ and $\mathbf{y}$ is computed as:

$$PCC(\mathbf{x}, \mathbf{y}) = \frac{\sum_{k=1}^{m}(x_k - \mu_\mathbf{x})(y_k - \mu_\mathbf{y})}{\sqrt{\sum_{k=1}^{m}(x_k - \mu_\mathbf{x})^2 \sum_{k=1}^{m}(y_k - \mu_\mathbf{y})^2}} \tag{2.21}$$

where

$$\mu_\mathbf{x} = \frac{1}{m}\sum_{i=1}^{m} x_i, \qquad \mu_\mathbf{y} = \frac{1}{m}\sum_{i=1}^{m} y_i \tag{2.22}$$

The value computed by $PCC(\mathbf{x}, \mathbf{y})$ is in the range $[-1, 1]$. Clustering algorithms usually employ the distance between input feature vectors, computed as:

$$d(\mathbf{x}, \mathbf{y}) = 1 - PCC(\mathbf{x}, \mathbf{y}) \tag{2.23}$$

such that the distance is always non-negative and the more similar the input vectors are, the smaller the distance between them is.

Clustering algorithms can be divided into the following categories:

- **hierarchical clustering** (Figure 2.10)
    - starts with a single cluster then subdivides into smaller clusters
    - exhibits a hierarchical relationship between the clusters
    - number of clusters not needed in advance

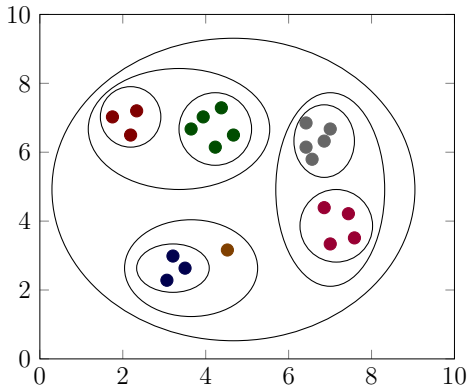


Figure 2.10: Hierarchical Clustering: Clusters exhibit a relational order with its subclusters.

- **partitioning clustering** (Figure 2.11)
  - ∗ divides the dataset into a specified number of clusters and adjust the membership of the data points to each cluster
  - ∗ does not exhibit hierarchical relationship between these clusters
  - ∗ requires the number of clusters to be known in advance

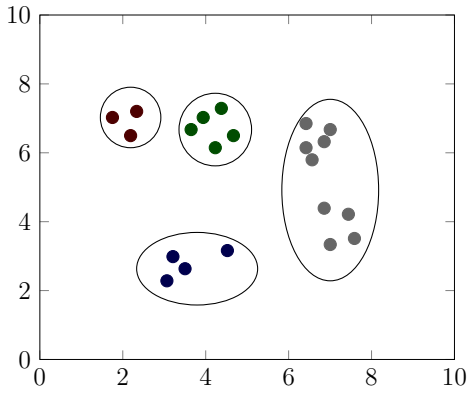


Figure 2.11: Partitioning Clustering: Datasets are strictly divided into separate clusters.

Due to the different approaches taken by these methods, my project will implement clustering algorithms based on both hierarchical clustering and partitioning clustering.

2.3 Rejected approaches

An additional NN architecture that was considered due to its widespread use is the convolutional neural network (CNN). However, since CNNs are, in practice,

employed on spatially-structured data (such as images), it was deemed to be inappropriate for the type and structure of the epigenetic data used in this project.

2.4 Requirements analysis

The background theory was important for understanding not only the inner workings of the chosen NN architectures and clustering algorithms, but also the high-level layout of the entire project. This section identifies the scope of the project through the requirements analysis.

The requirements were divided into functional and non-functional, as illustrated in Table 2.1. The functional requirements identify the modules of the project, while the non-functional requirements specify the criteria for correct behaviour of the supervised and unsupervised machine learning algorithm developed.

Requirement Description	Priority	Risk	Difficulty
Functional Requirements			
Comparative evaluation of NN Models	High	Medium	High
MLP implementation	High	Medium	Medium
RNN implementation	High	Medium	High
Epigenetic data clustering	Medium	Medium	Medium
SNN implementation	High	High	High
Epigenetic data processing	High	Medium	Medium
Synthetic data generation	Medium	Medium	Medium
Hyperparameter tuning	Medium	Low	Low
Non-Functional Requirements			
MLP achieves the expected accuracy on synthetic data	Medium	Low	Low
RNN achieves the expected accuracy on synthetic data	Medium	Low	Medium
SNN achieves the expected accuracy on synthetic data	Medium	Low	Medium
Clustering algorithm correctly finds clusters in synthetic data	Medium	Low	Low

Table 2.1: Description of high-level requirements for a successful project

After analysing the dependencies in the project (illustrated in Figure 2.12), a modularised implementation approach was adopted. This decision is motivated by the fact that the NN architectures can be independently implemented and tested. Nevertheless, a common interface is needed for the three models to allow integration with the comparative evaluation.

Additional modules such as those for hyperparameter tuning and clustering were also implemented and tested separately, then integrated with the NN models.

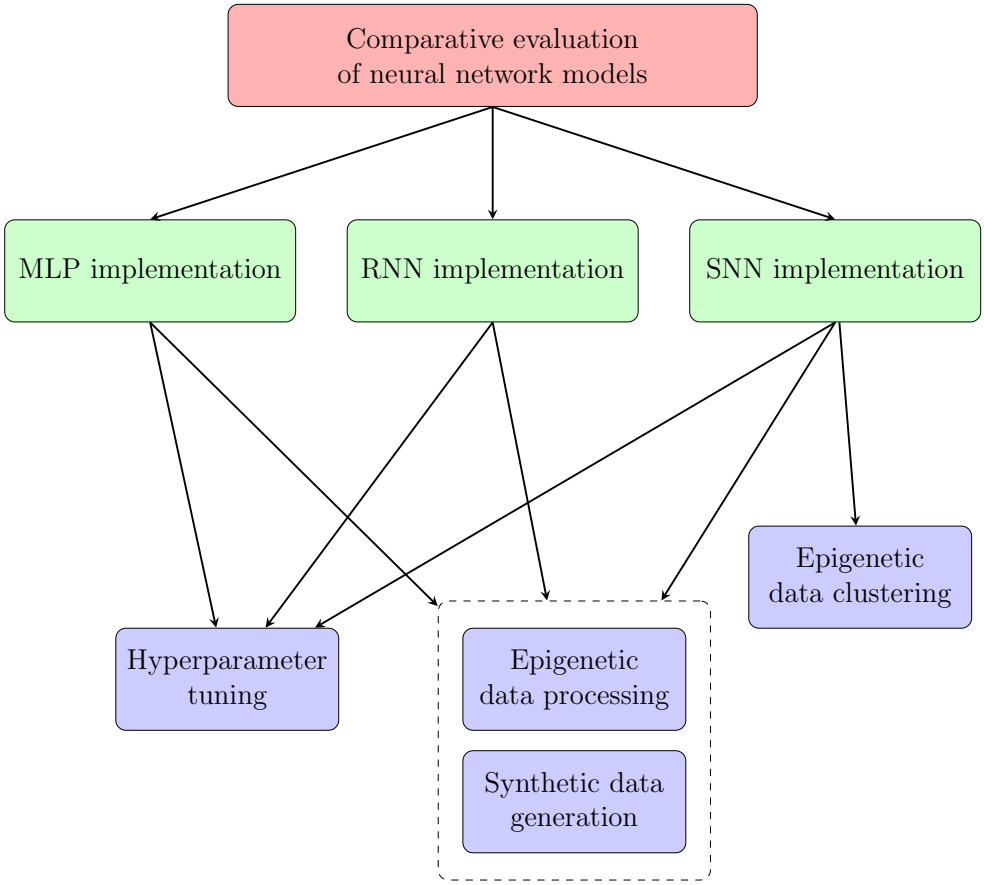


Figure 2.12: Directed Acyclic Graph of the dependencies in the project. $x \to y$ indicates that x depends on y.

2.5 Choice of tools

2.5.1 Machine learning library

The fundamental purpose of this project is to develop architectures for different types of NNs and evaluate their performance. Whilst the NN models could have been implemented from scratch, re-writing elementary operations (e.g. backpropagation), which have already been implemented and optimised by many libraries, detracts from the primary focus of the project: the fine-tuning of the high-level structure of these models. The use of an existing library therefore represents an appropriate choice for this project.

TensorFlow is a library that performs numerical computations on multidimensional arrays represented as `Tensor` objects. The NN models are described as dataflow graphs that perform operations on tensors.

The choice of TensorFlow was motivated by two reasons: the library is highly optimised for commonly used models such as the MLP and RNN, and it also provides the correct level of abstraction to build new NN models, such as the SNN.

2.5.2 Programming Language

TensorFlow provides a Python interface to a highly optimised C++ back-end. TensorFlow also has a C++ API, but it is not as complete and well-documented as the Python API.

While there is a range of programming languages that can be used for machine learning, Python offers the best balance of features as seen in Table 2.2. Python represents a good choice due to its concise syntax, and it allows for the development of the project in an object-oriented fashion with well defined classes and interactions between them. Python also provides flexibility and support for processing datasets and evaluating the NN models through libraries such as ***NumPy*** and ***SciPy***.

Comparison of programming languages				
	Python	**C++**	**Java**	**R**
Code readability	✓✓	✓	✓	✓
Concise syntax	✓	✓	×	✓
Ease of data handling	✓	×	×	✓
Object-oriented	✓	✓✓	✓✓	✓
TensorFlow support	✓✓	✓	×	×

Table 2.2: Comparison of common programming languages for machine learning.

The ***unittest*** framework was used for testing correct functionality of the code.

2.5.3 Development Environment

The project was implemented and tested on my laptop (2.6 GHz Intel Core i5 with 8GB RAM running Ubuntu). For the evaluation, Nvidia Titan X GPUs were used to speed up the training process of the NNs.

PyCharm was chosen as the IDE due to its integrated version control support, refactoring and debugging tools. The use of ***git*** for version control allowed me to: track key changes in the project, to roll-back if necessary, and to try out different strategies in developing the NN architectures and clustering algorithms.

2.5.4 Backup strategy

An effective backup strategy is required to reduce the risk of any software, hardware or user errors. The git repository for the project implementation was pushed to ***GitHub*** whenever a stable commit was made. Moreover, the repositories were synced with ***Google Drive*** and manual backups were made to an ***External HDD***.

2.6 Initial experience

At the beginning of the project, my knowledge of Python was intermediate, having used it before only for the group project in Part IB.

However, I had no previous experience with TensorFlow. This required me to become acquainted with the Python API, to learn how the library performs the computations on tensors and how NN architectures can be defined and optimised as data-flow graphs.

In terms of the theoretical background, the Part IB Artificial Intelligence course provided me with the necessary knowledge to understand how MLPs work and how they can be trained. Additional preparation was needed to understand the RNN and SNN, but also to choose the most appropriate architecture and regularisation techniques for each type of NN.

2.7 Software engineering techniques

2.7.1 Development model

The ***Iterative Development Model*** [17] was used in the implementation of my project as part of an Agile development methodology. An initial prototype was implemented for each module of the project, which was then iteratively refined. The purpose of the iterative refinements was to improve the performance of the models implemented or to add more functionality.

2.7.2 Testing

The project was thoroughly tested; unit tests were written for black-box testing, input sanitisation testing and integration testing.

The NNs and clustering algorithms were tested using synthetically generated datasets whose statistical properties can be fully controlled. The synthetic datasets were created algorithmically and specific gene patterns were introduced into them. The output of the NNs was compared against the expected accuracy on the synthetic data. The clustering algorithms were also validated to ensure identification of the correct structure in the synthetic dataset.

2.8 Summary

The Preparation chapter discussed the theory behind the machine learning models implemented in the project, identified the major components of the project through the requirements analysis and explored the software engineering techniques employed to ensure the success of the project.

Chapter 3

Implementation

This chapter describes the implementation of the MLP, RNN and SNN, the clustering algorithms, as well as the necessary data pre-processing performed to train the models and obtain the evaluation metrics.

Instead of analysing the low-level implementation details, the focus of this chapter is geared towards the high-level modules and the choices made in the design of the NN architectures and clustering algorithms.

Although this project in particular is not real-time or safety critical, good software engineering practices are nevertheless crucial; such a project could easily be part of large scale systems under high load. Consequently, the entire implementation was approached with certain key strategies in mind. For instance, time efficiency was achieved through judicious use of data structures and appropriate algorithms. Modularisation and code reuse as well as ensuring well-defined interactions between modules were essential for managing the project's complexity, supported by substantive documentation and use of Python's concise syntax.

3.1 Overall structure of the project

3.1.1 Nested cross-validation

The implementation of the project was predominantly focussed on the evaluation of the models. A brief overview of how the epigenetic data flows through the system will be covered to justify the design choices for the implementation of the NNs, but also for the pre-processing of the epigenetic data.

Figure 3.1 and Table 3.1 illustrate how the epigenetic data are used for nested cross-validation (CV): a process that firstly determines the best hyperparameters for training the NN, and then computes the evaluation metrics.

A particular benefit of this set-up is that it allows an unbiased way to simultaneously estimate hyperparameters, perform training and evaluate, while using the entirety of the available dataset. Being able to train and test on the entire dataset is particularly important in epigenetics, where the number of examples is limited. Since epigenetic datasets are imbalanced, stratified cross-validation is required, to ensure that similar proportions of each class are included in each fold.

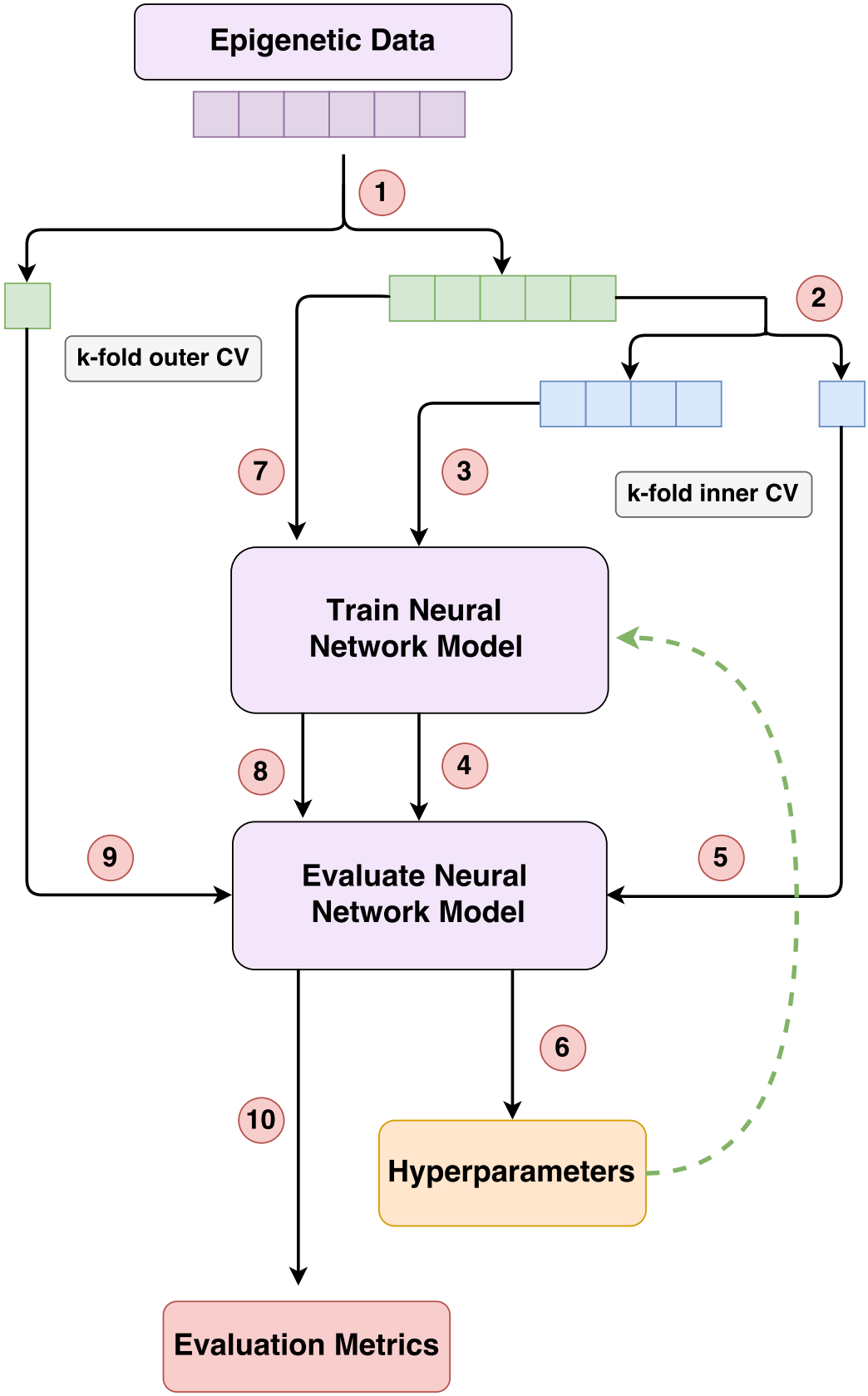


Figure 3.1: Flow of epigenetic data during nested cross-validation. Steps described in Table 3.1.

Nested Cross Validation		
Step	**Evaluation** (Outer CV)	**Hyperparameter Tuning** (Inner CV)
1	Split dataset into k_1 outer folds.	
2		Split $k_1 - 1$ of outer folds into k_2 inner folds.
3		Use $k_2 - 1$ of inner folds to train NN.
4		The NN computes optimal weights and biases that minimise loss on training data.
5		Use remaining inner fold to evaluate model and compute performance metrics.
6		For selected possible values of hyperparameters, repeat steps 3-5 until each of the k_2 inner folds has been used as test data. At the end of this process, the most suitable hyperparameters have been selected.
7	Use $k_1 - 1$ of the outer folds to train the model and set the hyperparameters to the ones determined by the Inner CV.	
8	The NN optimises the weights and biases for the current training dataset.	
9	Use the remaining outer fold to evaluate the NN.	
10	Steps 2-9 are repeated until each of the k_1 splits has been used as test data yielding the overall performance metrics.	

Table 3.1: Description of the steps involved in nested CV

3.1.2 Data structures

The NNs are implemented as TensorFlow computational graphs. Several data structures are needed to represent the data flow and the structure of the computational graph built:

- input data to the NN,
- weights and biases associated with each architecture,
- TensorFlow computational graphs for each NN.

The input to the NN for each mini-batch is represented as a rank 2 tensor, where each row consists of the input features for a training example, and the number of rows is equal to the size of the mini-batch.

The weights and biases are described as TensorFlow `Variable` objects that are trainable and maintain a persistent state across evaluations of the TensorFlow graph. In terms of in-memory representation, `Variable` objects consist of mutable handles to buffers storing tensors. The weights for each layer are represented as tensors of rank 2 and of type `float32`, while the biases are tensors of rank 1 and of type `float32`. The low precision used for the arithmetic computations is sufficient for training and also yields faster hardware computations. [5]

The TensorFlow computational graph is described by its operations on `Tensor` objects. A node is added to the graph for each operation used. Edges between the nodes represent the data flow of the tensors through the graph. The advantage of this representation is that it provides an easy way of understanding dependencies between the units of the computational model. Moreover, it clarifies which parts of the graph perform sequential computation and which may be parallelised.

3.2 Epigenetic data

3.2.1 Structure of the epigenetic datasets used

The NNs developed in the project will be used on the inference tasks described in Table 3.2.

Epigenetic Datasets		
	Dataset 1	**Dataset 2**
Classification Task	Predict embryonic development stages	Diagnose patients with cancer
Input data	1-dimensional array consisting of gene expression levels	1-dimensional array consisting of gene expression levels and/or DNA methylation levels.
Output	One out of 7 classes representing embryonic development stages	One out of 2 classes representing whether the patient has cancer
Dataset size	90 examples	590 examples

Table 3.2: Description of epigenetic datasets used

These two datasets were chosen carefully due to their unique characteristics. An example in the embryonic development datasets consists of thousands of gene expression levels. Clustering will be used to study the underlying structure of the data and to obtain partitions for the superlayered architecture.

Conversely, the information to diagnose cancer for a patient has two different types of input data: gene expression levels and DNA methylation levels. The interactions between these two different modalities will be studied.

3.2.2 Epigenetic data class implementation

The structure of epigenetic datasets and the data flow described in Table 3.1 justifies the UML class hierarchy represented in Figure 3.2. Each class supports a method for retrieving the k-fold datasets for nested CV.

The high number of available gene expression levels in the embryo development dataset requires pre-processing to select the most relevant genes. Pre-processing was achieved in the following steps:

- normalise gene expression levels and compute entropy of the genes across different classes. The number of genes to be selected and minimum required entropy are specified at instantiation.
- cluster selected genes (`EmbryoDevelopmentDataWithClusters`) to obtain the different cluster inputs for the SNN.
- select a single cluster (`EmbryoDevelopmentDataWithSingleCluster`) to evaluate the behaviour of the MLP and RNN on more refined data.

The structure of the dataset for cancer patients allows for a clear separation into different data modalities: gene expression levels and DNA methylation levels. These modalities are combined in the `CancerPatientsData` class, separated in the `CancerPatientsDataWithModalities` class or only one of them is used in the `CancerPatientsDataDNAMethylationLevels` class.

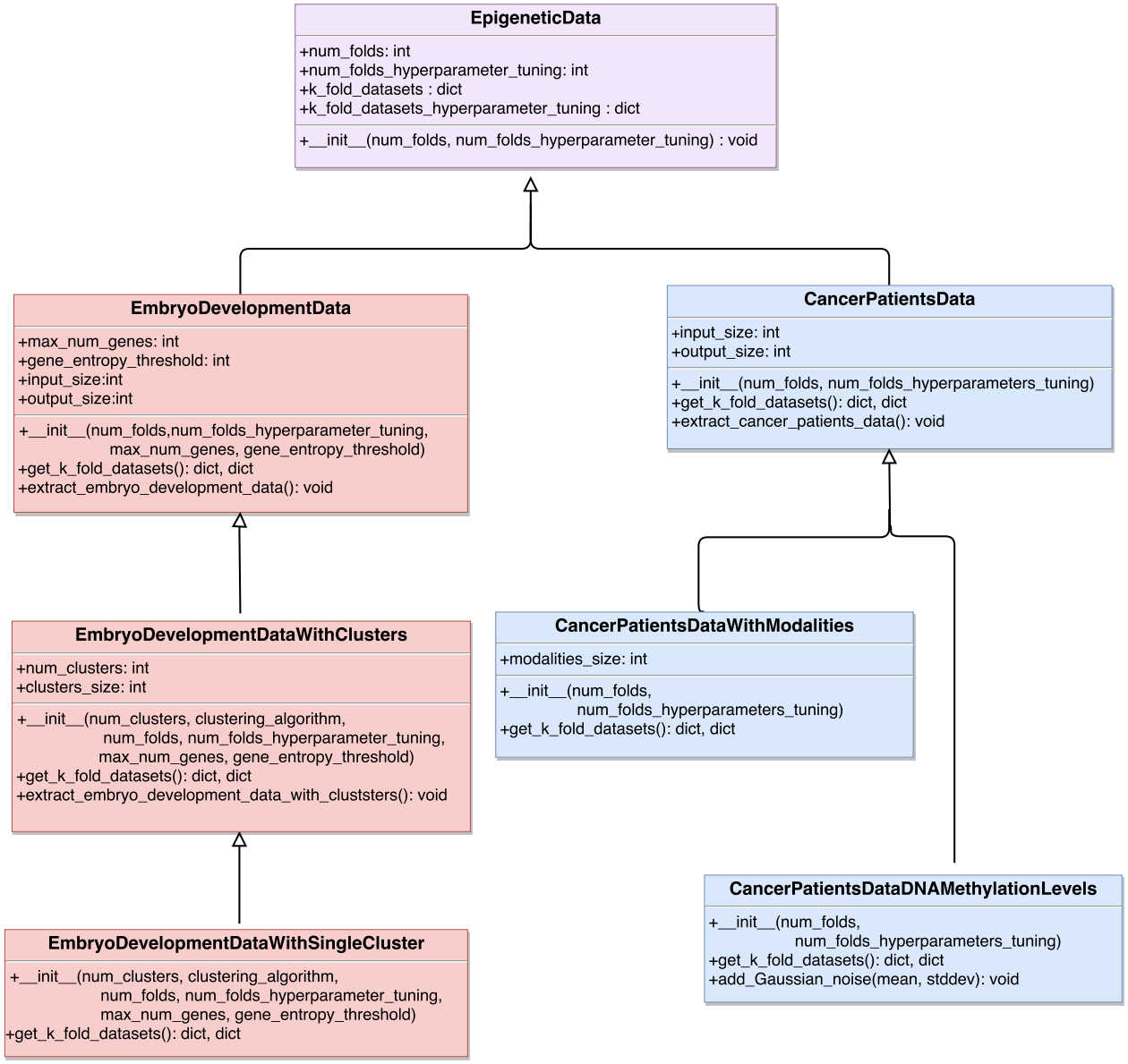


Figure 3.2: UML class inheritance diagram for epigenetic data.

3.3 Multilayer perceptron

This section gives details about the specific architecture chosen for the MLP and describes its TensorFlow implementation. Regularisation techniques used to account for data sparsity and class imbalance are also discussed.

3.3.1 MLP architecture

In practice, an MLP architecture consists of an expansion phase, where the number of neurons in the hidden layers increases, and a reduction phase, where this number decreases. This approach allows for features in the input data to be combined in the expansion phase and then for the most important features to be selected to

determine the class in the reduction phase.

However, having an expansion phase increases the number of parameters the network needs to optimise by $\mathcal{O}(nm)$ for a transition from n to m features. This approach is not appropriate for epigenetic data, where datasets are small and there is insufficient data to adequately optimise a large number of parameters. The MLP architecture I found most appropriate for my project consists only of the reduction phase, where the features in the epigenetic data are refined over four hidden layers. Figure 3.3 represents the high-level overview of this architecture, where the connections between layers illustrate the data flow through the network.

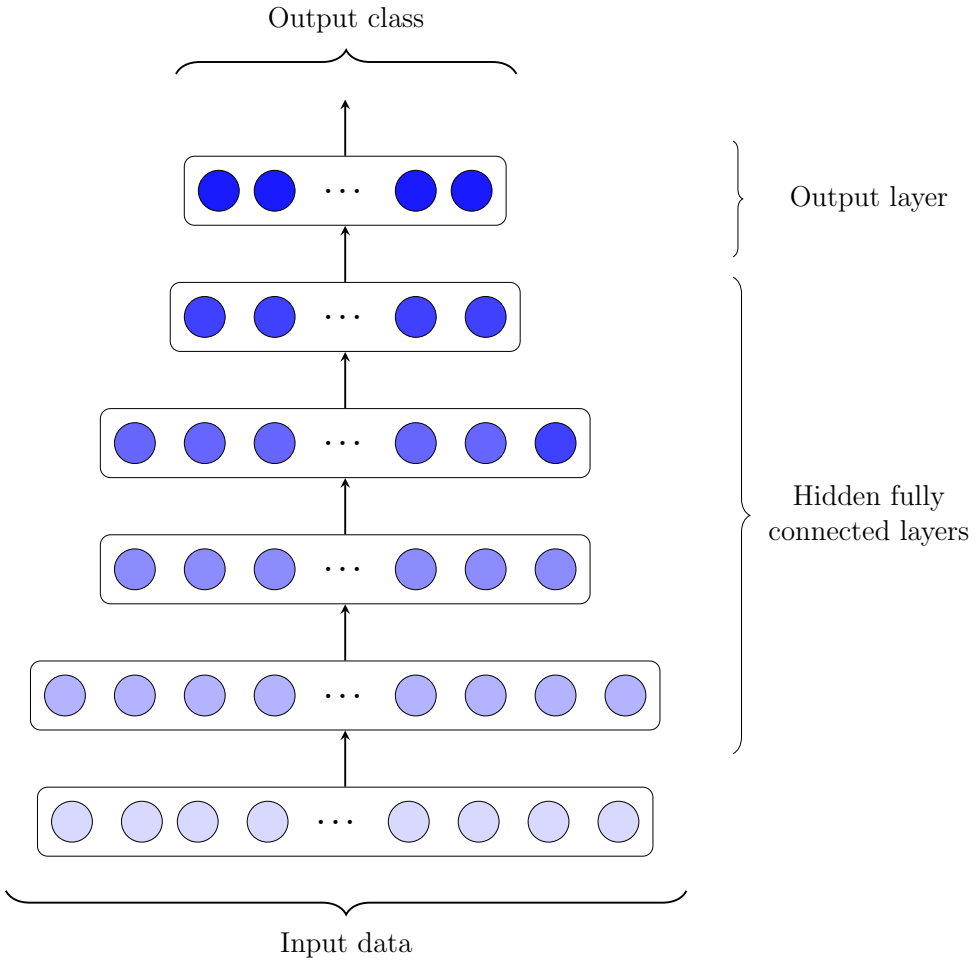


Figure 3.3: MLP architecture and data flow.

The **ReLU** function was chosen as the **activation function** for the neurons in the MLP due to its linear non-saturating form. Moreover, by having a steep gradient, ReLU accelerates the convergence of the training algorithm. [16]

3.3.2 MLP parameters initialisation

The initial value of the parameters determines the starting point of the optimisation performed during training. Improper initialisation can cause the training algorithm to diverge or to get stuck in a suboptimal local minima.

A naïve initialisation for the weight involves assigning a constant value to all of them. In this situation the input to each neuron in a hidden layer will have the same value which will lead to the gradients computed by the network to be the same. This poses a severe limitation on the network's learning capacity.

Good initialisation methods assign the weights to unique random values. When using the ReLU activation function, He initialisation [10] gives the best results by choosing an initial value that helps the signal reach the deeper layers of the network.

He initialisation involves choosing weights for each layer from a Gaussian distribution with:

$$\mu = 0, \qquad \sigma^2 = \frac{2}{n_{\text{in}}} \tag{3.1}$$

where n_{in} represents the number of neurons that go into the layer. The biases are usually initialised to zero, since the necessary initial randomness is already ensured by the weights.

3.3.3 MLP class overview

The class design for the MLP (Figure 3.4) is tightly coupled with the TensorFlow implementation of an NN and to the nested CV design described in §3.1.1.

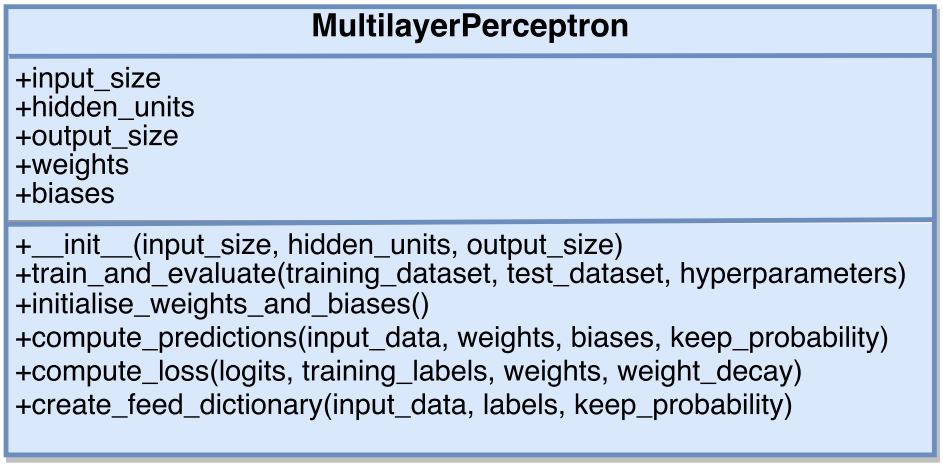


Figure 3.4: MultilayerPerceptron class.

The common interface implemented by each model consists of the `train_and_evaluate` method which is used from within the outer and inner folds of the nested CV module. This method builds the TensorFlow graph, optimises it

using the training data, evaluates its performance on the test data and returns the evaluation metrics.

The operations performed by the `train_and_evaluate` method are similar across the NN models implemented. Therefore, I shall describe the method in detail for the MLP and then only highlight the differences for the RNN and SNN.

3.3.3.1 Build the TensorFlow graph

The TensorFlow graph is built by adding nodes describing the operations performed at each layer through the network and by specifying the data flow through these layers.

The main nodes of the computational graph are the:

- *weights and biases* for each layer,
- *nodes performing the operations* at each layer,
- *loss node* that computes the cross-entropy loss between the predictions and the true labels,
- *optimiser node* that updates the weights and biases in order to minimise the training loss,
- *placeholder nodes* that are used to represent the input tensors to the computational graph.

During training, the data are fed to the graph through the placeholder nodes.

3.3.3.2 Train the TensorFlow graph in a Session object

TensorFlow uses a lazy language paradigm, where the computational graph nodes have no numerical value until they have been evaluated, thus separating the definition of the graph from its execution. The environment in which a computational graph is evaluated is encapsulated in a `Session` object.

After the `Session` object is instantiated, the variables in the graph are initialised and the TensorFlow graph is executed for some number of steps until the weights and biases are optimised. At each step, a mini-batch is selected from the training dataset and a 'feed dictionary' is built to map the placeholder nodes to the mini-batch data. During the execution of the graph, TensorFlow automatically computes the gradients of the training loss with respect to the weights and biases. Then, the evaluation of the optimiser node updates the parameters to minimise the loss.

3.3.3.3 Evaluating neural network models in TensorFlow

After the TensorFlow graph has been optimised in the `Session` object, it can be used the make predictions on test data.

During evaluation, input placeholders are mapped to the test data and the predictions of the NN are obtained by evaluating the output node.

3.3.4 Operations performed by MLP layers

The operations performed by each hidden layer can be represented by the nodes of the computational graph illustrated in Figure 3.5. This design is refined by adding the regularisation techniques described in §3.3.8, §3.3.7.

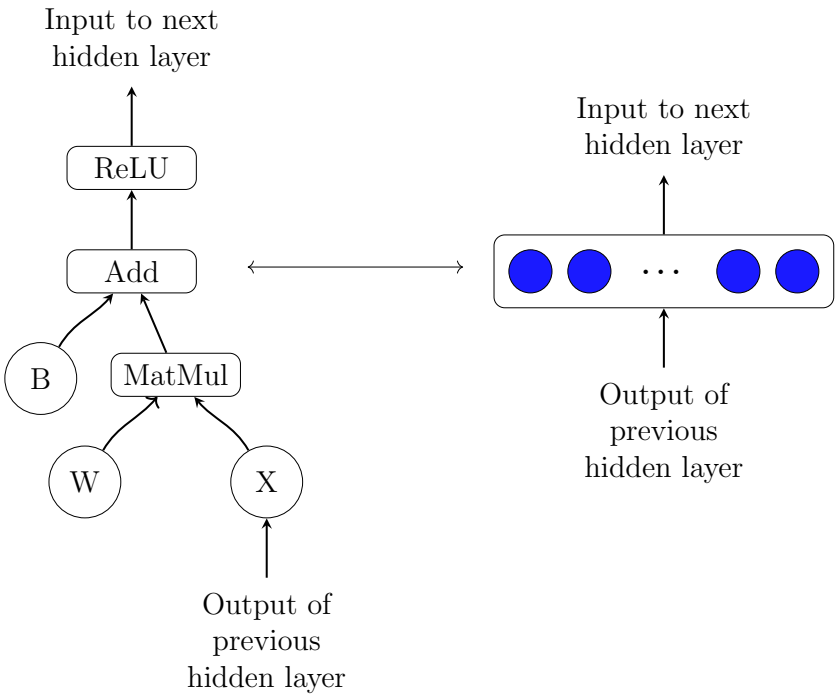


Figure 3.5: The nodes added to the data-flow graph (*left*) to perform the operations of each hidden layer (*right*).

The output layer (Figure 3.6) computes the weighted sum of its input and adds the bias. This gives the output `logits`[1]. The softmax node is added to the graph to transforms `logits` into a probability distribution over the possible classes.

[1]The name for `logits` comes from the logit function $f(x) = \log(\frac{x}{1-x})$, which represents the inverse of the logistic function.

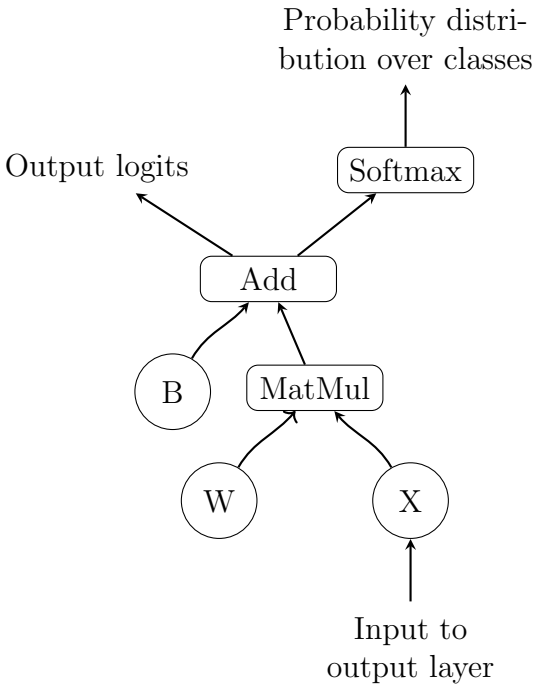


Figure 3.6: Nodes added to computational graph to perform the operations of the output layer.

3.3.5 Loss of MLP

The loss node computes the cross-entropy loss between the output distribution and one-hot ground-truth labels, evaluated after each training iteration. Weight decay is used as a regularisation technique that controls model complexity by penalising large weights.

Weight decay is achieved by changing the loss function as follows:

$$\mathcal{L} = -\sum_{i=1}^{k} y_i \log\left(y_i'\right) + \frac{\lambda}{2}\|\mathbf{w}\| \tag{3.2}$$

where $\|\mathbf{w}\|$ represents the L_2 norm of the weights in the network and λ is the weight decay **hyperparameter** that needs to be optimised.

Weight decay ensures that the model only uses the weights it needs. Furthermore, the model becomes smoother since the output will change more slowly with changes in input.

3.3.6 Optimiser

The optimiser used to train the MLP is mini-batch gradient descent, which performs the operations described in §2.1.5. The learning rate is a hyperparameter that establishes the step size used to update the gradient.

3.3.7 Batch normalisation

During training, the distributions of inputs to each hidden layer may have significant discrepancies, a problem known as internal covariate shift. Batch normalisation (BatchNorm) addresses this problem by normalising the input values to every layer across each mini-batch. [13]

Let $\{\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_m\}$ be the input elements to layer L in the network over the mini-batch $\mathcal{B}$. BatchNorm shifts and scales each $\mathbf{x}_k$ by the mean and variance of the inputs over $\mathcal{B}$ respectively. Each normalised input $\hat{\mathbf{x}}_k$ is computed as follows:

$$\hat{\mathbf{x}}_k = \frac{\mathbf{x}_k - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}} + \epsilon}} \tag{3.3}$$

where ϵ is a small, non-negative constant added for numerical stability and $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ are computed using sample mini-batch statistics, i.e.

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{x}_i, \qquad \sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} \left(\mathbf{x_i} - \mu_{\mathcal{B}}\right)^2 \tag{3.4}$$

Simply applying this re-standardisation decreases the learning capabilities of the network by restricting the input range to a hidden layer. BatchNorm allows the model to fall back to the original values by learning a scale (γ) and a shift (β) parameter for $\hat{\mathbf{x}}_k$. The final output of BatchNorm is:

$$\mathbf{z_k} = \gamma \hat{\mathbf{x}}_k + \beta \tag{3.5}$$

During testing, the normalised inputs are computed using the population statistics, the trained scale and shift parameters.

In terms of TensorFlow implementation, BatchNorm is applied to the inputs to activation functions in each layer as represented in Figure 3.7.

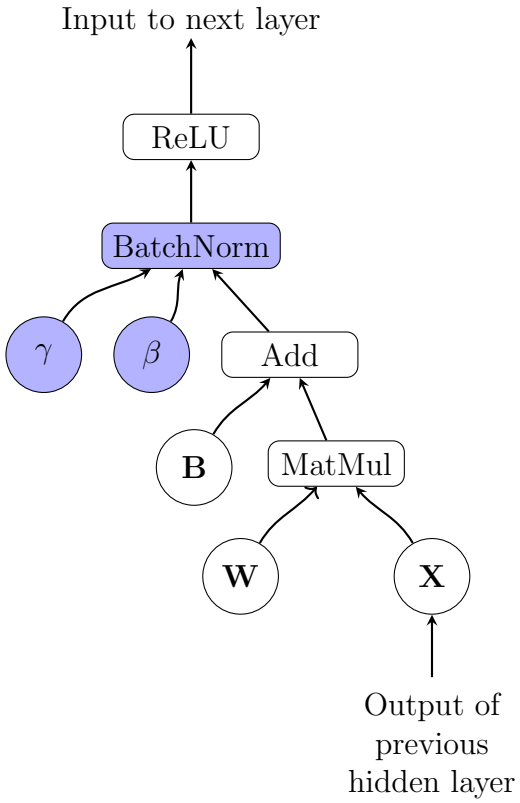


Figure 3.7: Adding the BatchNorm node.

3.3.8 Dropout

Dropout [29] is a regularisation technique applied during training and it involves randomly removing neurons from the network. The reason for using dropout is to ensure that the network does not rely on a small number of neurons. Dropout can be interpreted as sampling the network for each training iteration, such that only weights and biases of the sampled neurons are updated during that iteration.

Figure 3.8 illustrates the idea of applying dropout to an MLP. During training, a neuron in a given layer is kept with some *keep probability* k_p. Since dropout is not applied during testing, the corresponding weights to the next layer need to be scaled by $1/k_p$ so that the expected value over the outputs of the hidden layer remains unchanged.

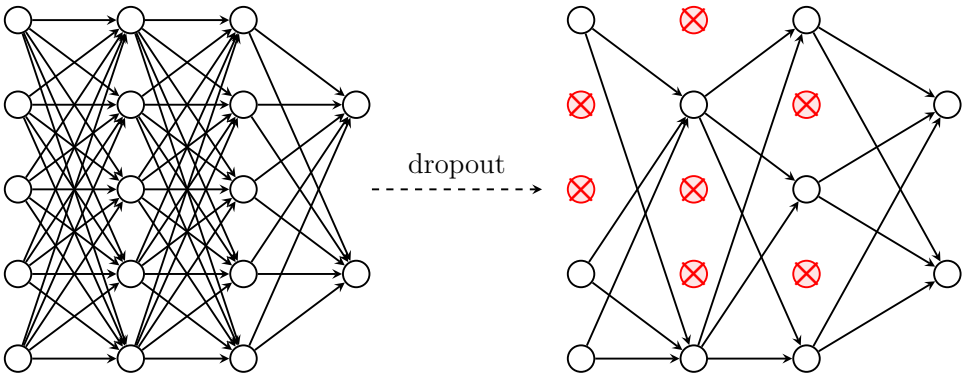


Figure 3.8: *Left*: The architecture of the NN before dropout is applied. *Right*: The architecture after dropout is applied and neurons are removed during training with probability $1 - k_p$.

In TensorFlow, dropout is applied after the activations of a given layer are computed as represented in Figure 3.9.

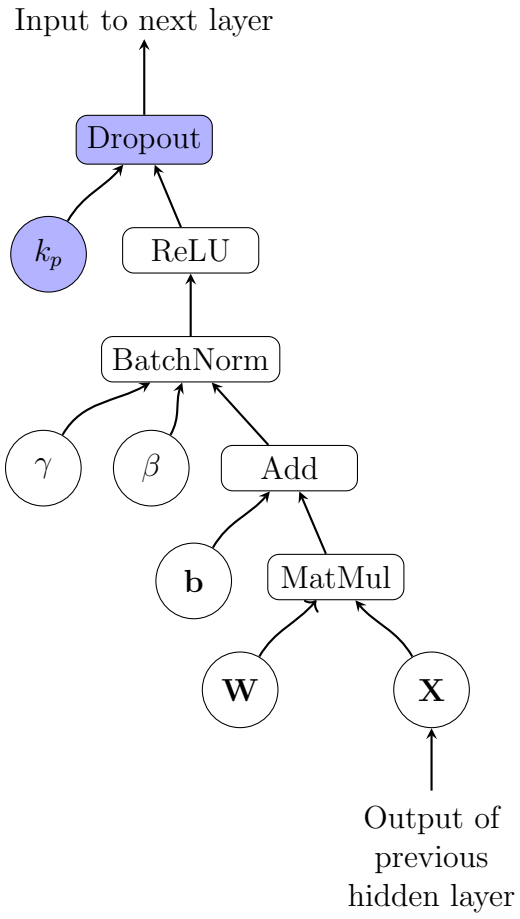


Figure 3.9: Adding the dropout node.

The value of k_p is optimised as a **hyperparameter**. Since dropout is only applied during training, a placeholder is created for k_p: during training, the placeholder maps to the optimised hyperparameter, while during testing it maps to 1.0.

3.4 Recurrent neural network

This section describes the implementation of the RNN and its applicability to the analysis of epigenetic data in a sequential manner.

3.4.1 RNN Architecture

Having a single LSTM cell will not produce a model with enough capacity to extract the complex features in epigenetic data. Therefore, for the RNN architecture, I decided to use two stacked LSTM cells (LSTM_1 and LSTM_2) arranged such that the output of LSTM_1 is given as input to LSTM_2.

Since the parameters are shared in the LSTM unit, it makes sense to increase the number of neurons in LSTM_2 to extract more features.

At timestep T, the features computed by LSTM_2 are given as input to two fully connected layers that combine them to compute the output class. Since the fully connected layers add a large number of parameters to the model, the number of neurons in the fully connected layers will decrease with each newly added layer. The architecture of the fully connected layer is similar to the architecture of the MLP. The high-level view of the RNN architecture is illustrated in Figure 3.10.

The LSTM units use the **logistic** and **hyperbolic tangent activation functions** as described in §2.1.7.1. The fully connected layers, however, perform the same computations as the hidden layers of the MLP. Consequently, it is a reasonable choice to use the **ReLU activation function** for these layers.

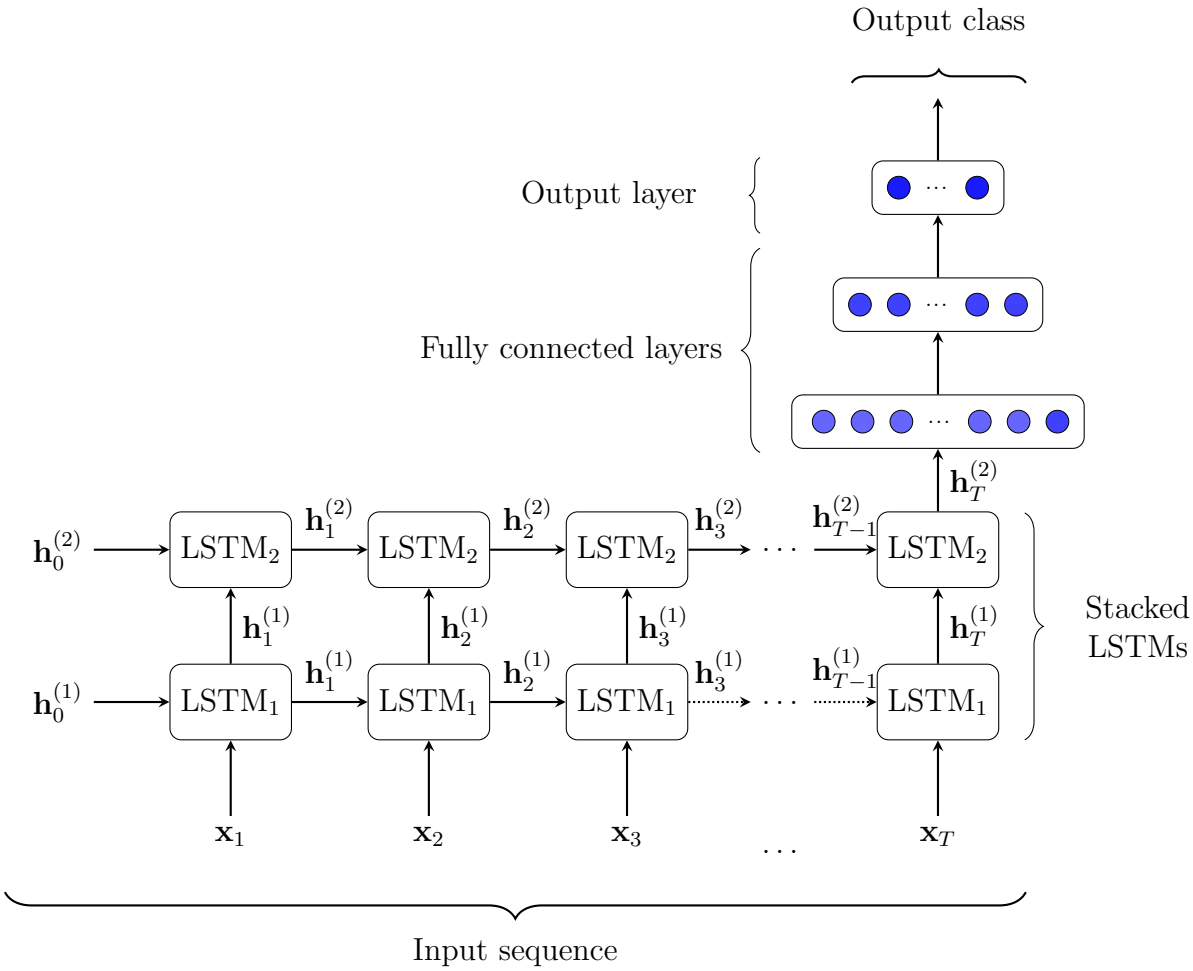


Figure 3.10: Data flow in the RNN.

3.4.2 RNN parameters initialisation

The weights associated with each LSTM unit are initialised using Xavier initialisation,[8] which is a similar to He initialisation, but adapts the variance of the distribution to make it more appropriate for the logistic and tanh activation functions. Under this scheme, the weights for each RNN unit are chosen from a normal distribution with:

$$\mu = 0 \qquad , \quad \sigma^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}} \tag{3.6}$$

where μ is the mean, σ^2 is the variance of the distribution, n_{in} is the number of input elements and n_{out} is the number of outputs computed by the unit.

The forget bias needs to be initialised to a vector of ones to establish gradient flow and to encourage long-term dependencies at the onset of training; other biases in the LSTM unit are initialised to zero.

3.4.3 RNN class overview

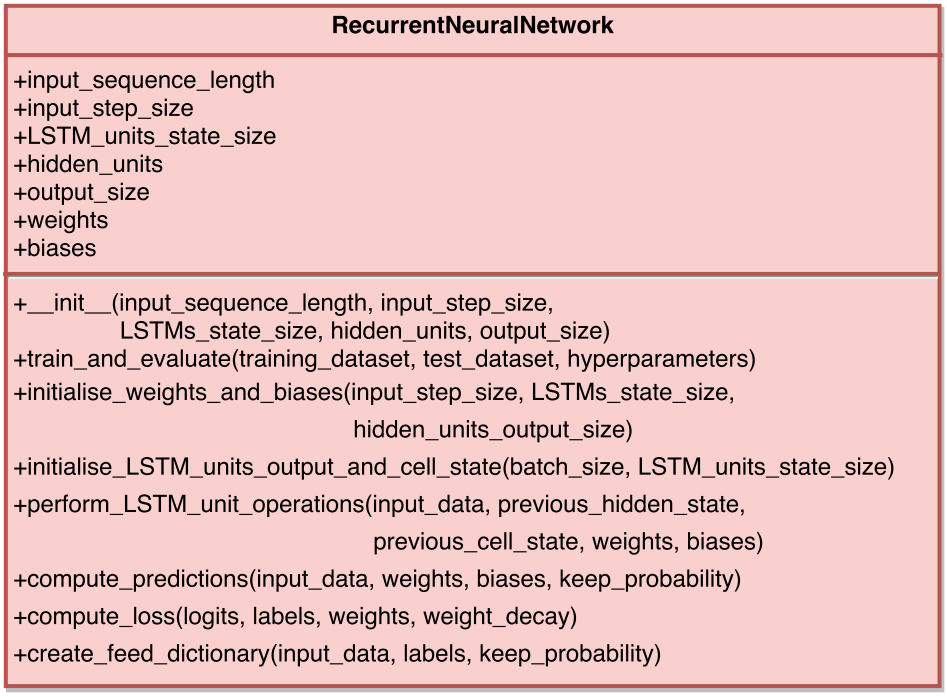


Figure 3.11: RecurrentNeuralNetwork class

The structure of the `RecurrentNeuralNetwork` class is similar to the `MultilayerPerceptron` class. The RNN consists of stacked LSTM units and fully connected layers, as described in §3.4.1. The method used to build, optimise and evaluate the TensorFlow graph is `train_and_evaluate`.

The operations performed by the fully connected layers and their initialisation are the same as for the MLP layers, already described in §3.3.2, §3.3.4. The cross-entropy loss is used again, with weight decay applied as a form of regularisation (see §3.3.5).

3.4.4 Operations performed by RNN layers

At each timestep, the input data are analysed by LSTM_1 and then the output of LSTM_1 is given as input to LSTM_2. Each LSTM unit has an initial cell state $\mathbf{c}_0$ and an initial hidden state $\mathbf{h}_0$, which are set to `Tensor` objects consisting of zeros.

The computational graph for the LSTM units is built in a `for`-loop such that at timestep t:

1. LSTM_1 receives input $\mathbf{x}_t$, applies gated operations to $\mathbf{x}_t$, $\mathbf{h}_{t-1}^{(1)}$ and $\mathbf{c}_{t-1}^{(1)}$ and computes $\mathbf{c}_t^{(1)}$ and $\mathbf{h}_t^{(1)}$.

2. LSTM_2 receives input $\mathbf{h}_t^{(1)}$, applies gated operations to $\mathbf{h}_t^{(1)}$, $\mathbf{c}_{t-1}^{(2)}$ and $\mathbf{h}_{t-1}^{(2)}$ and computes $\mathbf{c}_t^{(2)}$ and $\mathbf{h}_t^{(2)}$.

These operations t are illustrated in Figure 3.12. The output of the LSTM_2 after T timesteps, h_T, is given as input to the fully connected layers.

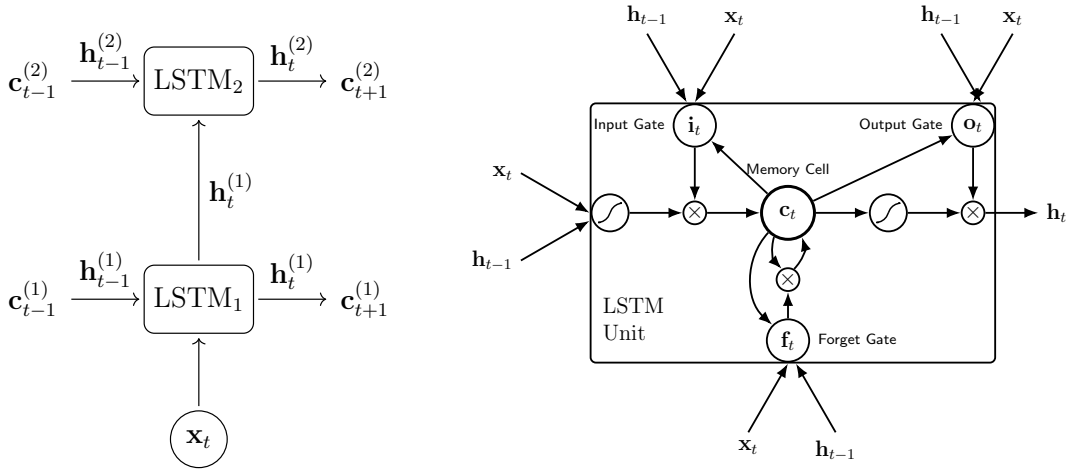


Figure 3.12: *Left*: Data flow of $\mathbf{x}_t$, $\mathbf{c}^{(1)}$, $\mathbf{c}^{(2)}$, $\mathbf{h}^{(1)}$ and $\mathbf{h}^{(2)}$ through the stacked LSTMs. *Right:* Data flow of $\mathbf{x}_t$, $\mathbf{h}_{t-1}$ and $\mathbf{c}_t$ through a single LSTM unit.

3.4.5 Optimiser

When using the gradient descent optimiser, the learning rate hyperparameter needs to be assigned. In particular, setting the learning rate for the RNN is even more difficult than for the MLP because the weights and biases within an RNN unit are shared and therefore are independently and simultaneously modified several times per iteration, once for every step in the training sequences.

Therefore, for the RNN, I decided to use instead the **Adam optimiser** that computes an adaptive learning rate for each parameter. The intuition for this approach is that the loss function can be sensitive to some directions of the gradient in the parameter space, but insensitive to others. Consequently, the optimisation algorithm should update the loss function in the correct direction such that it reaches a (desirable) local minima. An explanation of how Adam optimiser performs these updates is given in Appendix B.

While Adam optimiser still requires an initial value for the learning rate, the training process of the network will be less sensitive to it.

3.4.6 Regularisation techniques

For the LSTM layers, dropout is used on the hidden-to-hidden connection, as illustrated in Figure 3.13. This regularisation is important for epigenetic data since it is very longitudinal. Applying dropout on the recurrent connections has been shown to cause instability,[35] and therefore dropout will not be used on there.

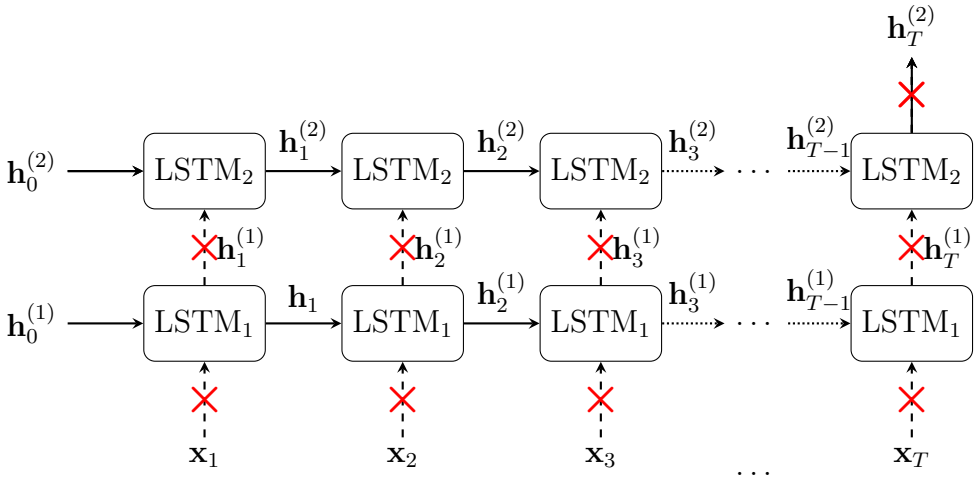


Figure 3.13: Dropout applied to the LSTM units.

While a modification of BatchNorm suitable for recurrent networks was recently introduced,[3] it often exhibits extreme sensitivity to hyperparameters, and consequently was deemed too unstable for use in this project.

3.5 Clustering

For the purpose of my project, clustering is used as a pre-processing step to exploit the structure of epigenetic data. The clustering algorithms are defined in separate Python modules, where each module consists of the main algorithm and any auxiliary methods needed.

Clustering is used to identify groups of co-expressed genes. The input consists of a mapping from each gene to its expression levels while the output maps each gene to the corresponding cluster.

3.5.1 Hierarchical Clustering

Hierarchical clustering was used to determine the most appropriate number of clusters to partition the data into, while build a dendrogram of the data points. The dendrogram is plotted using the `scipy` library, and allows for the hierarchies between different clusters to be visualised, as shown in Figure 3.14.

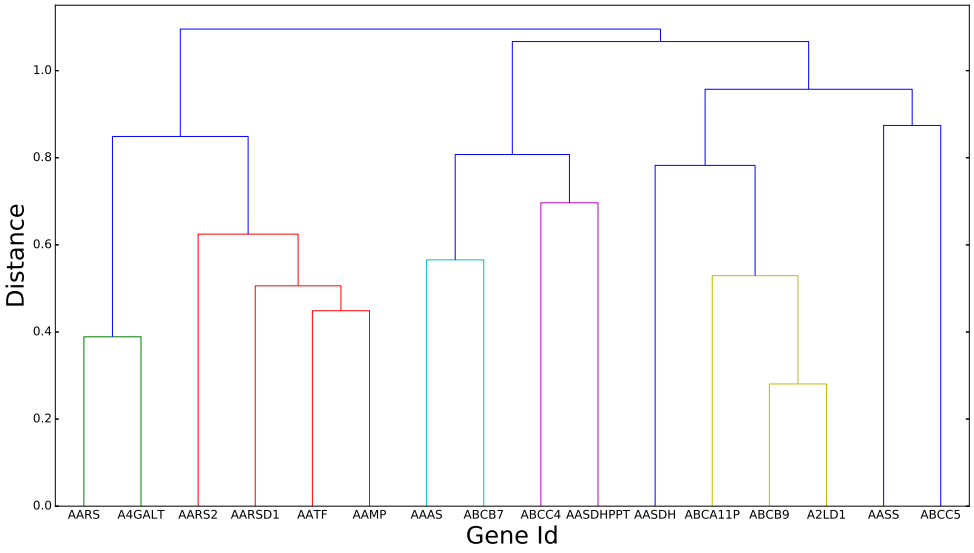


Figure 3.14: Typical dendrogram obtained for a sample of 16 genes.

The clusters can be determined by plotting a horizontal line on the dendrogram and determining the points of intersection, as illustrated in Figure 3.15. Moving the line through the dendrogram will result in a different number of clusters.

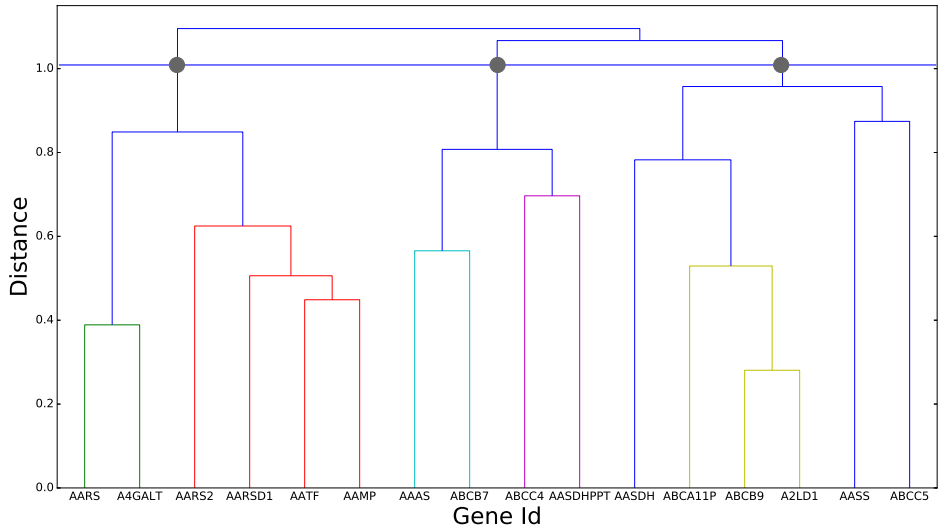


Figure 3.15: Visualisation of clusters from a dendrogram.

Since the design of the SNN requires only two input clusters, the adopted approach was to use the dendrogram to determine the best number of clusters to split the data into and then apply the hierarchical clustering algorithm until the specified

number of clusters is formed. The clusters with the largest number of data points are then used as inputs to the SNN.

The main **data structure** used by hierarchical clustering is the distance matrix D, which at each step t retains the distances between clusters that are formed by step t.

Initially, gene $\mathbf{g}_i$ forms cluster $\mathcal{C}_i$, and D consists of the distance between the expression levels of genes $\mathbf{g}_i$ and $\mathbf{g}_j$ which is computed using the Pearson Correlation Coefficient:

$$D_{ij} = 1 - PCC(\mathbf{g}_i, \mathbf{g}_j) \tag{3.7}$$

Then, the following steps are repeated until the desired number of clusters is reached:

1. Select the clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ with the smallest distance between them; this is achieved by finding the minimum value in D.

2. The genes in $\mathcal{C}_i$ and $\mathcal{C}_j$ are combined into a single cluster $\mathcal{C}_k$, which is added to D.

3. The distance between the newly formed cluster $\mathcal{C}_k$ and the other clusters is computed by taking the average between the distance from $\mathcal{C}_i$ to the other clusters and the distance from $\mathcal{C}_j$ to the other clusters. For an arbitrary existing cluster $\mathcal{C}_l$, D_{kl} may be derived as:

$$D_{kl} = \frac{D_{il} + D_{jl}}{2} \tag{3.8}$$

4. Clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ are removed from D.

When computing D_{kl}, several alternative approaches can be used such as:

- the minimum value between D_{il} and D_{jl}:

$$D_{kl} = \min(D_{il}, D_{jl}) \tag{3.9}$$

- or the weighted average:

$$D_{kl} = \frac{D_{il} \cdot |i| + D_{jl} \cdot |j|}{|i| + |j|} \tag{3.10}$$

  where $|i|$ and $|j|$ represent the number of elements in clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ respectively.

After an initial trial with all these methods, the method observed to give the most balanced clusters involved the use of the average of the two distances.

3.5.2 k-means clustering

k-means clustering is a partitioning method and requires the number of input clusters k to be specified as a parameter. Let $\mathcal{D} = \{\mathbf{g}_1, \mathbf{g}_2, ..., \mathbf{g}_n\}$ be the initial dataset, where $\mathbf{g}_i \in \mathbb{R}^m$ is regarded as a point in an m-dimensional data space.

An algorithm for the k-means clustering problem seeks to compute the centre $\mathbf{c}_i$ for each cluster $\mathcal{C}_i$ such that the pairwise squared distance between each data point assigned to $\mathcal{C}_i$ and $\mathbf{c}_i$ is minimised. However, this is known to be NP-hard even for $k = 2$ in Euclidean space,[1] and therefore an approximate heuristic known as Lloyd's algorithm will be employed. *PCC* is used again in computing the distance between the data points.

The main **data structure** used for k-means clustering is a map between each cluster ID and its centre, represented as an m-dimensional array. The algorithm goes through several iterations that update these centres until the cluster assignments no longer change.

The steps involved in the implementation of Lloyd's algorithm are:

1. Initialise centres to k random data points from $\mathcal{D}$ (Forgy initialisation). This initialisation method is more likely to spread the centres throughout the m-dimensional space.

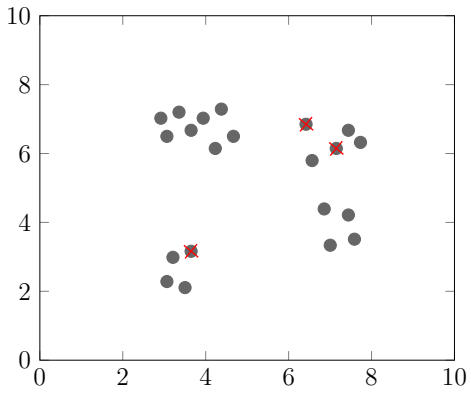


Figure 3.16: Initialisation step in Lloyd's algorithm. The data points are represented in two dimensional space for clarity.

2. Each data point is then assigned to the closest centre to form clusters: $\mathbf{g}_i$ becomes part of $\mathcal{C}_t$, iff

$$d(\mathbf{g}_i, \mathbf{c}_t) = \min_{\forall p \in \{1,...,k\}} d(\mathbf{g}_i, \mathbf{c}_p) \tag{3.11}$$

where $d(i, j)$ is the distance between points i and j.

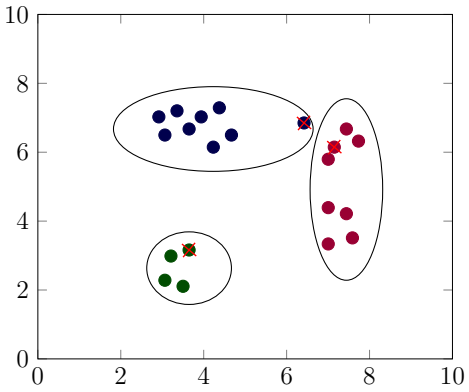


Figure 3.17: The cluster formed after each data point is assignment to the closest centre.

3. The position of each cluster centre $\mathbf{c}_t$ is re-computed as the centre of gravity[1] of the data points assigned to $\mathcal{C}_t$:

$$\mathbf{c}_t = \frac{1}{|\mathcal{C}_t|} \sum_{\mathbf{g}_i \in \mathcal{C}_t} \mathbf{g}_i \tag{3.12}$$

where $|\mathcal{C}_t|$ are the number of data points assigned to cluster t.

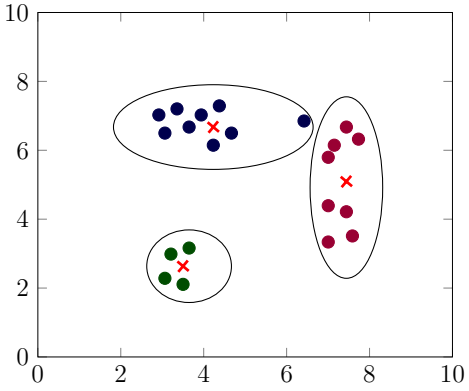


Figure 3.18: Position of new cluster centres.

4. Steps 2 and 3 are repeated until cluster assignments no longer change, i.e. the algorithm has converged.

[1]The centre of gravity minimises the pairwise squared distance between $\mathbf{c}_t$ each data point in $\mathcal{C}_t$.

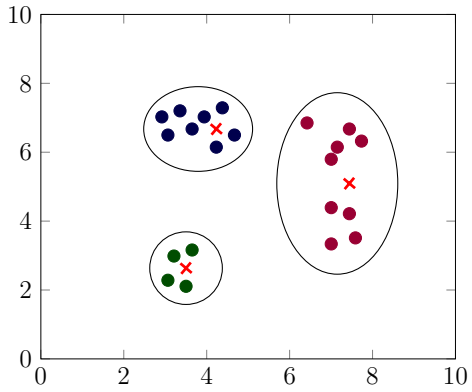


Figure 3.19: Clusters obtained after the k-means clustering algorithm converges.

The best number of clusters to partition the dataset into is determined from the dendrogram plotted for hierarchical clustering. The main difference between k-means and hierarchical clustering is that cluster assignments for hierarchical clustering are fixed; once two clusters are combined, the cluster assignments no longer change. However, with k-means the data points can still move between clusters through the iterations. The two methods are likely to produce different clustering results, which will be analysed in the Evaluation chapter.

3.6 Superlayered Neural Network

The third and final architecture implemented is the SNN which explores the interactions between different partitions of the epigenetic data.

3.6.1 SNN Architecture

The structure of the model was designed by taking into account the trade-off between the number of parameters in the network and the size of the dataset that can be used to optimise them.

The model consists of two separate feedforward superlayers. To be able to make a comparison between the MLP and SNN, I decided to use the same number of hidden layers in a superlayer as the number of hidden layers in the MLP. To ensure that data can flow freely through the entire network, the superlayers are cross-connected. The cross-connections were added in a way that preserves symmetry.

The features learnt by the superlayers are eventually concatenated, and two fully connected layers were added to operate on them simultaneously and produce the network's predictions.

Figure 3.20 illustrates the SNN architecture and the data flow through this network.

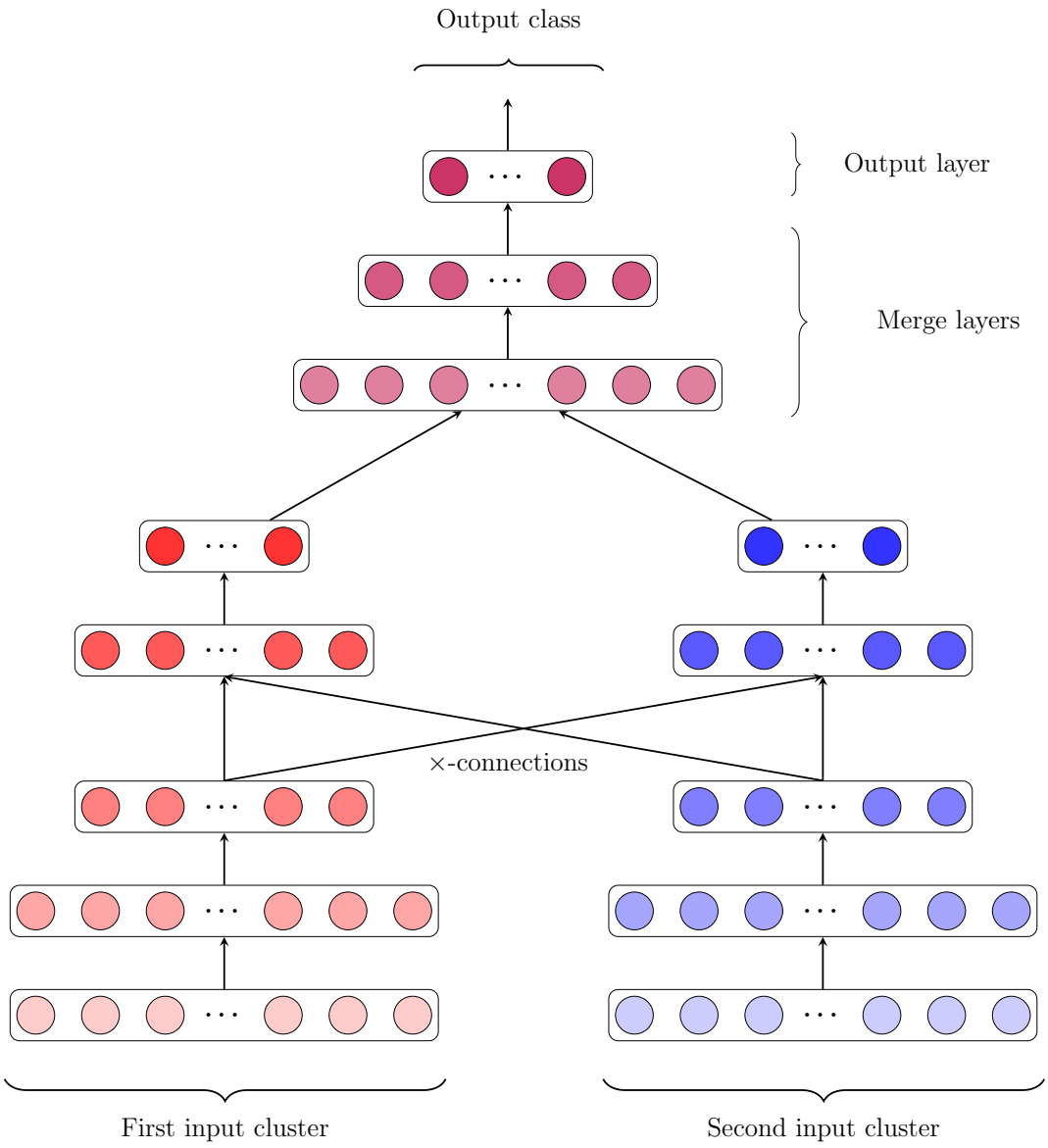


Figure 3.20: Data flow through the SNN

Initialisation, optimisation, and regularisation techniques applied by this model exactly match the ones for the MLP (already described in §3.3.2, §3.3.5, §3.3.6, §3.3.7, §3.3.8) and will therefore not receive particular attention here.

3.6.2 Superlayered Neural Network Class

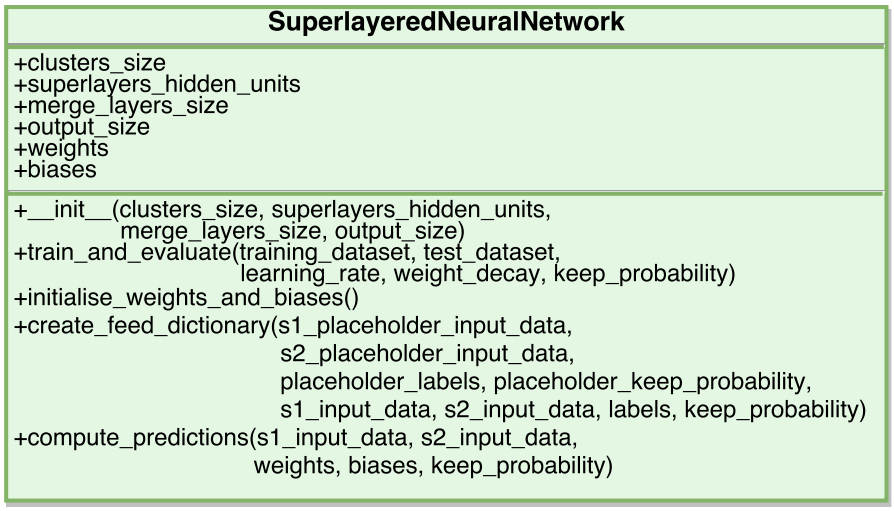


Figure 3.21: SuperlayeredNeuralNetwork class

The `SuperlayeredNeuralNetwork` class provides the same interface for training and evaluating the model. An important difference between the SNN architecture and the architectures described earlier is that the input data are separated into clusters, and each cluster is given as a separate input to each superlayer.

3.6.3 Operations performed by cross-connected layers

The third hidden layer in each superlayer has a cross-connection. Therefore, the input to such a layer consists of the output of the second layer in each superlayer multiplied by the corresponding weights. Figure 3.22 illustrates the operations performed by such a layer.

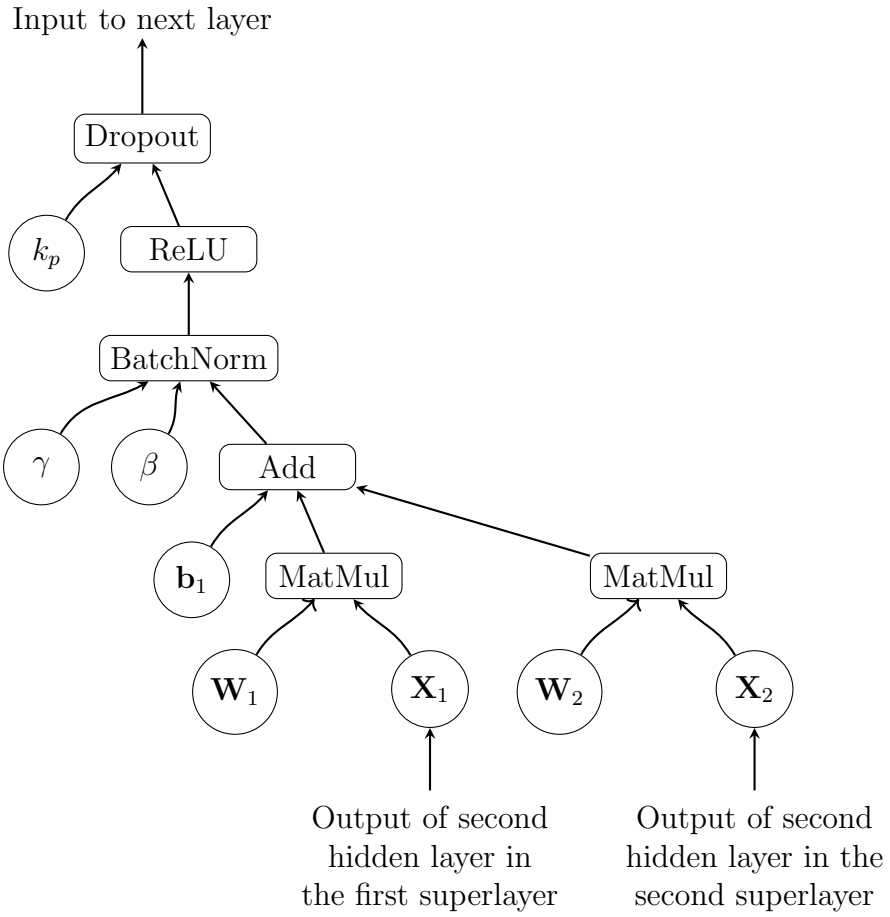


Figure 3.22: Operations performed by the layer in the first superlayer receiving a cross-connection.

3.7 Reflecting on the software development process

3.7.1 Iterative development

The Iterative Development Methodology chosen was particularly important for the implementation of the NNs. This methodology allowed for rapid trial and error of different techniques to improve performance.

For each architecture, a minimal viable working model was initially implemented which only built the corresponding data-flow graph. These models were observed to significantly overfit on the synthetic data and even diverge at times. Through each iteration, a regularisation technique was added and the corresponding change in performance was analysed to determine whether more regularisation was needed or whether the change was detrimental to the model. The chosen regularisation techniques (weight decay, dropout and BatchNorm) were all observed to be beneficial

to the performance of the models.

3.7.2 Timetable refinement

While careful planning was necessary in the initial project proposal (Appendix D) to enable the timely and successful implementation of the project, additional modules needed to be added to satisfy the requirements identified in Table 2.1.

In particular, the original proposal did not allocate time for proper implementation of the epigenetic data processing, synthetic data generation and hyperparameter tuning modules. Nevertheless, buffer slots in the timetable allowed for their implementation to be performed without introducing significant delay.

Figure 3.23 gives a more refined plan for the development of the project and also illustrates the period over which the modules were completed.

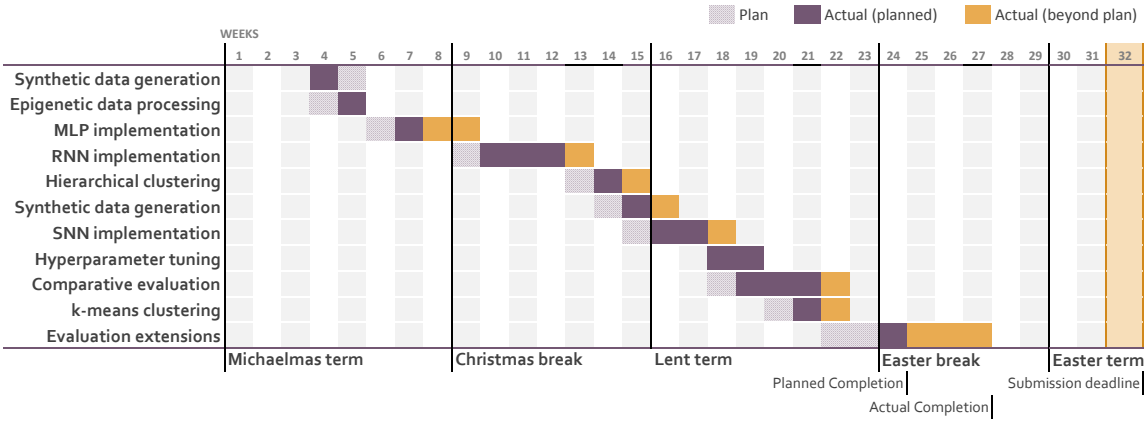


Figure 3.23: Gantt chart for the development process.

3.8 Summary

This chapter has illustrated the flow of epigenetic data through nested cross-validation, the implementation of the NN architectures as well as the clustering algorithms used for pre-processing the data. Additionally, several trade-offs were discussed for deciding on the architecture of each NN. The following chapter will thoroughly evaluate the models implemented.

Chapter 4

Evaluation

This chapter discusses how success criteria were achieved and exceeded, how the models implemented were tested against synthetic data and how different values for the hyperparameters affect the performance. Finally, the results obtained from comparative evaluation of the models on different datasets and under various conditions are illustrated and explained.

4.1 Success Criteria

The success criteria described in the project proposal (Appendix D) for the core project and extensions have all been met. This section highlights these achievements along with the corresponding sections detailing the work done. (✓) marks successful implementation.

Criterion 1: *Implement an MLP, RNN and SNN.* (✓)

The criterion was successfully achieved, as exemplified by §3.3.1, §3.4.1, §3.6.1.

Additionally, the most appropriate values for the hyperparameters were determined within the hyperparameter tuning module (§4.3), implemented as part of the nested CV.

Extension: *Implement an additional clustering algorithm.* (✓)

Two distinct clustering algorithms (hierarchical and k-means) were successfully implemented, as shown in §3.5.

Criterion 2: *Test correctness of models on synthetic data.* (✓)

Synthetic datasets were created as described in §4.2. Both the NNs and clustering algorithm have the expected behaviour on the synthetic data.

Criterion 3: *Evaluate models on epigenetic datasets and utilise good assessment metrics to account for the sparse and imbalanced data.* (✓)

The NNs were trained and evaluated using nested CV on two different types of datasets (§4.5, §4.6) using appropriate evaluation metrics (§4.4).

The following extensions were implemented to gain further insight into the performance of the models:

Extensions:

- *Evaluate the RNN and MLP with input obtained from a cluster.* (✓) (§4.6.1)
- *Compare clustering algorithms, based on their effect on the m performance.* (✓) (§4.6.1)
- *Assess robustness of the models to noise.* (✓) (Appendix C)

4.2 Evaluate models on synthetic data

Due to the inherently stochastic elements within NN training algorithms (such as mini-batch sampling and dropout), a deterministic test for enquiring whether the NNs are learning properly cannot be used.

One viable alternative involves exploiting carefully crafted synthetic datasets which allow the NN to be tested in a well-defined environment, with pre-determined patterns inserted into the dataset. The different structure of inputs to the MLP, RNN and SNN requires distinct artificial datasets to be created to test each model. Synthetic data are also used for determining whether the clustering algorithms are correctly partitioning the input data.

4.2.1 Synthetic Datasets for testing the MLP and RNN

The synthetic datasets for testing the MLP and RNN are created as follows:

- for each example in class C_i:
  - choose n elements from the distribution $\mathcal{N}(\mu, \sigma^2)$
  - randomly select k elements and shift their mean such that they have the distribution $\mathcal{N}(\mu + \Delta\mu, \sigma^2)$

The synthetic dataset used for testing consists of two classes C_1 and C_2, with 500 data points per class, out of which 100 were used for evaluation. When selecting $\mu = 0$, $\Delta\mu = 1$, $\sigma^2 = 0.5$ and $k = 1$, the probability of error is approximately 0.20[1] and the MLP obtains an 87% accuracy, while the RNN achieves an 82% accuracy.

Increasing $\Delta\mu$ and k will increase the intra-class variability. The models tested with $\Delta\mu = 1$ and $k = 16$ achieve perfect classification.

By achieving the expected accuracy on synthetic data, the MLP and RNN satisfy the non-functional requirements set for their successful implementation.

[1]The probability of error is determined from the overlapping area of the Normal distributions.

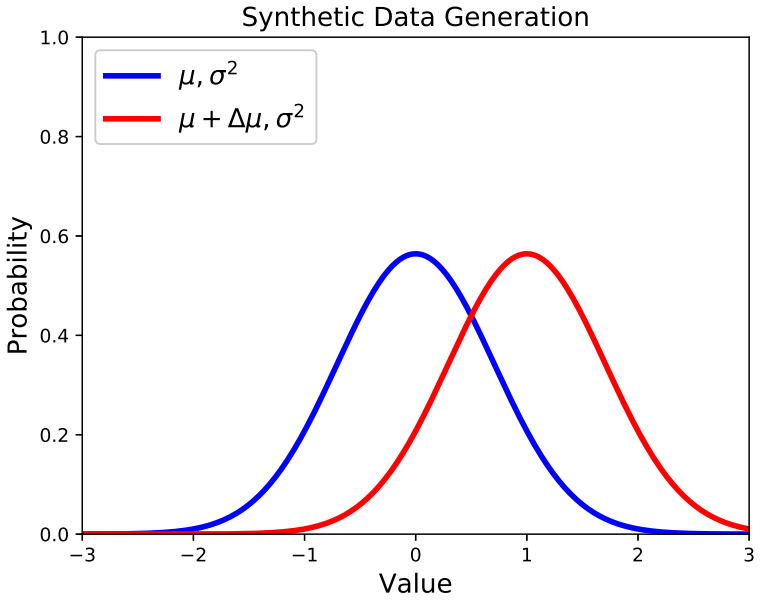


Figure 4.1: For each data point, select $n-k$ values from $\mathcal{N}(\mu, \sigma^2)$ and k values from $\mathcal{N}(\mu + \Delta\mu, \sigma^2)$

.

4.2.2 Synthetic Data for SNN and clustering algorithms

The artificial dataset for testing the SNN requires the input data to be separated into clusters. The following approach was taken for generating synthetic data for the SNN:

- for each example in class C_i:
  - ∗ randomly select k elements
  - ∗ simulate cluster i by:
    - ◇ choosing n elements from the probability distribution $\mathcal{N}(\mu_i, \sigma^2)$
    - ◇ shifting the mean of the k elements such that they have the distribution $\mathcal{N}(\mu_i + \Delta\mu, \sigma^2)$

The artificial dataset for testing the SNN also consists of 500 examples for each of the classes C_1 and C_2. By training the SNN on 400 examples and selecting $\sigma = 0.5$, $\Delta\mu = 1$, and $k = 1$, the probability of error is approximately 0.1 and the SNN achieves an accuracy of 94%. Similarly, setting $\Delta\mu = 2$ and $k = 16$ results in perfect classification.

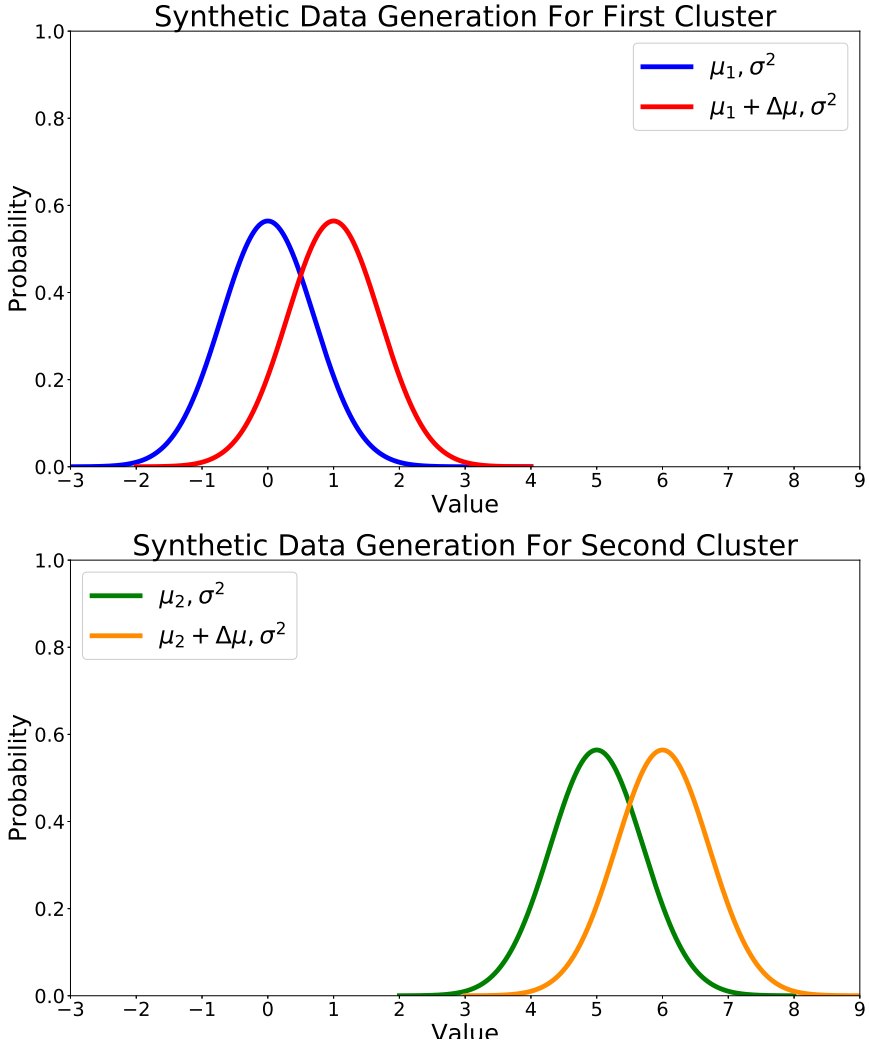


Figure 4.2: For each data point: form first cluster by taking $n - k$ values form $\mathcal{N}(\mu_1, \sigma^2)$ and k values from $\mathcal{N}(\mu_1 + \Delta\mu_1, \sigma^2)$ and form second cluster by taking $n - k$ values from $\mathcal{N}(\mu_2, \sigma^2)$ and k values from $\mathcal{N}(\mu_2 + \Delta\mu_2, \sigma^2)$.

.

The data points for testing the clustering algorithms are created as follows:

- create m data points of the form $\mathbf{x} = [x_1\, x_2]$, where $x_1 \sim \mathcal{N}(\mu_1, \sigma^2)$ and $x_2 \sim \mathcal{N}(\mu_2, \sigma^2)$
- create m data points of the form $\mathbf{y} = [y_1\, y_2]$, where $y_1 \sim \mathcal{N}(\mu_3, \sigma^2)$ and $y_2 \sim \mathcal{N}(\mu_4, \sigma^2)$

Both hierarchical and k-means clustering were able to partition the $2m$ data points into the appropriate clusters.

The results obtained using the SNN and clustering algorithms on the synthetic data satisfy the remaining non-functional requirements of the project for successful implementation of these models.

4.3 Hyperparameter Tuning

As discussed in §3.1.1, the project was developed such that nested stratified CV could be used for evaluation.

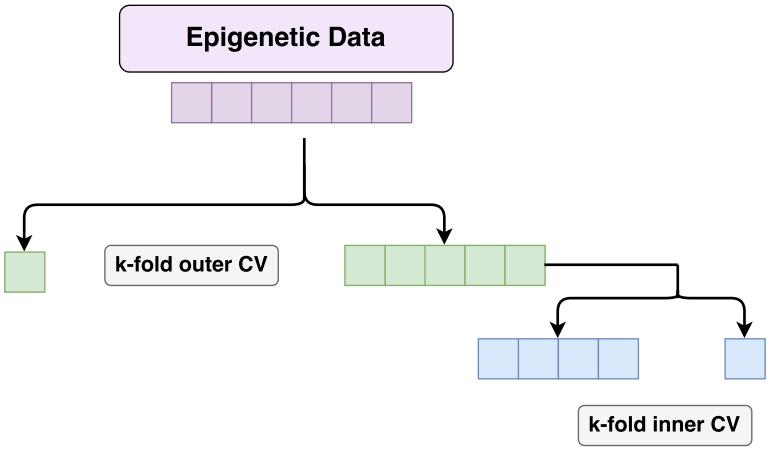


Figure 4.3: Splitting epigenetic data for nested CV.

The hyperparameters were determined within the inner CV, by testing the models over a wide range of possible values and settling on those that give the smallest error rate, computed as the ratio between the misclassified examples and the total number of examples. The hyperparameters optimised in this way are the learning rate, keep probability (used for dropout) and weight decay.

The following sections will explain how the value for each hyperparameter affects the error rate of a model. To aid these explanations, the error rate is plotted after 10-fold inner CV on the cancer patients dataset. The error bars represent the standard deviation of the error rate over the 10 folds.

4.3.1 Learning Rate

The learning rate decides by how much the direction of the gradient is followed at each step of gradient descent. A small learning rate might require a larger number of iterations to reach the minima, but it can give better results, by oscillating closer to this minimum point. While a large learning rate can lead to faster convergence, gradient descent might not be able to reach as close to the minima as when having a small learning rate, as shown in Figure 4.4.

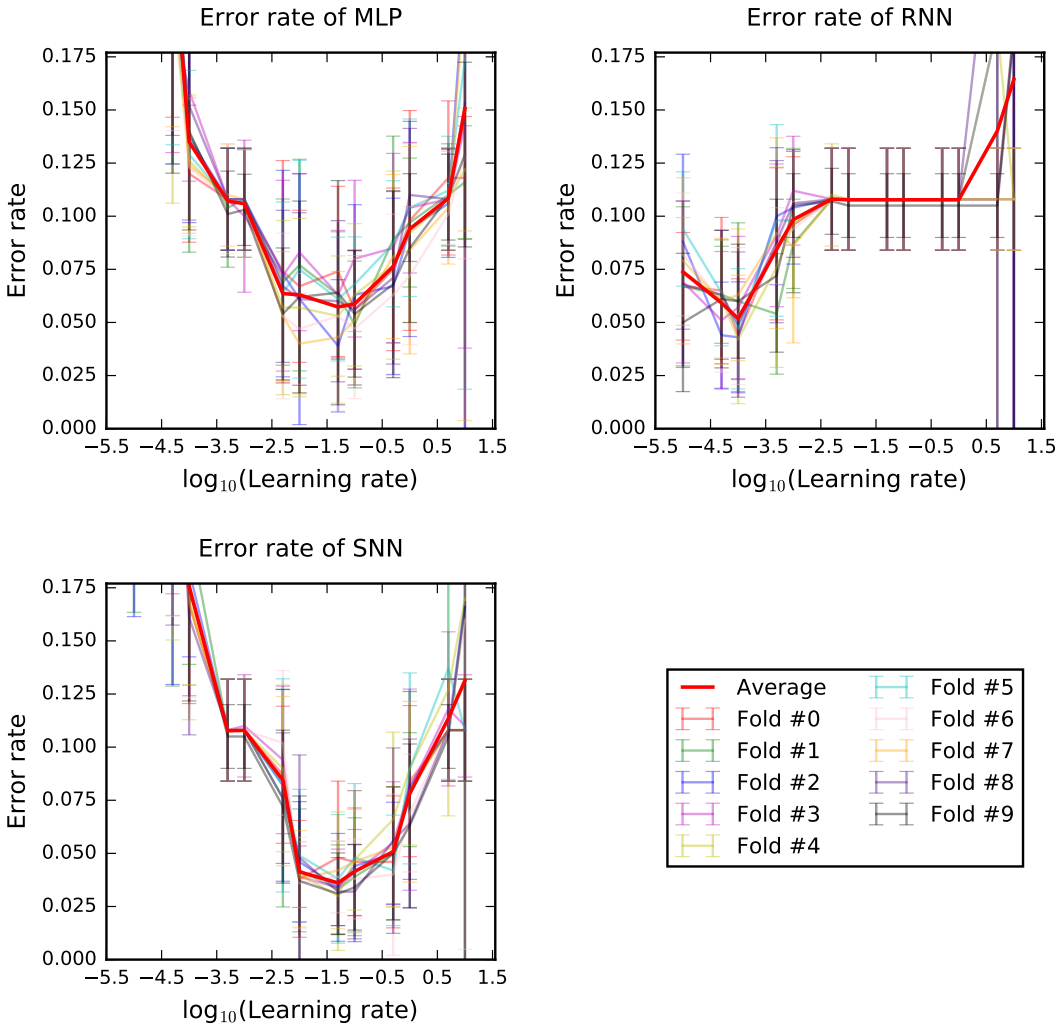


Figure 4.4: Overall trend of error rate with varying the learning rate.

4.3.2 Dropout

Dropout is used as a form of regularisation to ensure that the model does not heavily depend on specific neurons. The corresponding hyperparameter is the keep probability which has values in $[0, 1]$. A low value for the keep probability heavily reduces the learning capacity of the NN, while a high value might result in overfitting as observed in Figure 4.5.

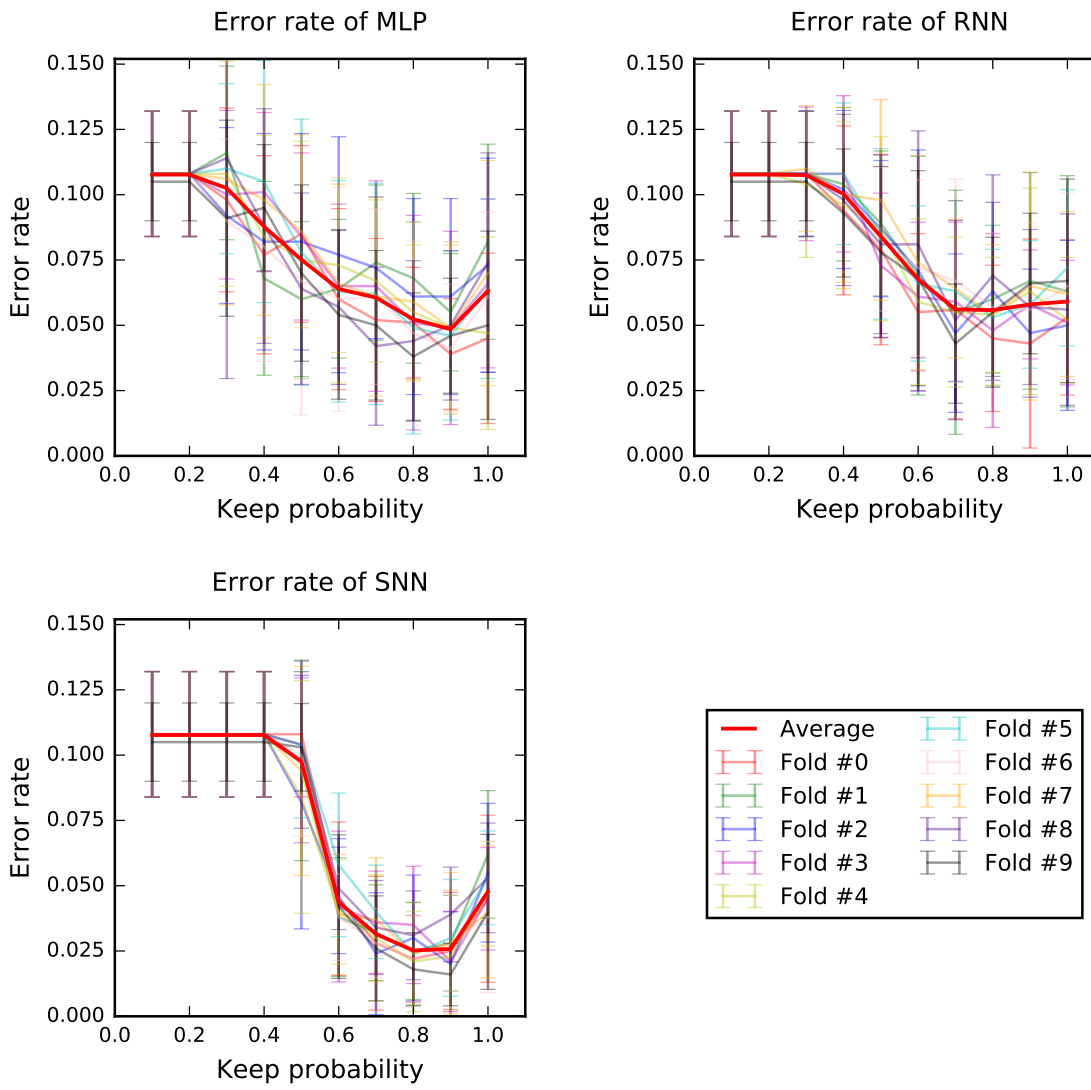


Figure 4.5: Overall trend of the error rate with varying keep probability.

4.3.3 Weight decay

Weight decay penalises large weights in the network. A high value for the weight decay can result in the network setting all the weights to zero, while a low value causes more pronounced overfitting, as the weights receive insufficient regularisation. Figure 4.6 illustrates the results.

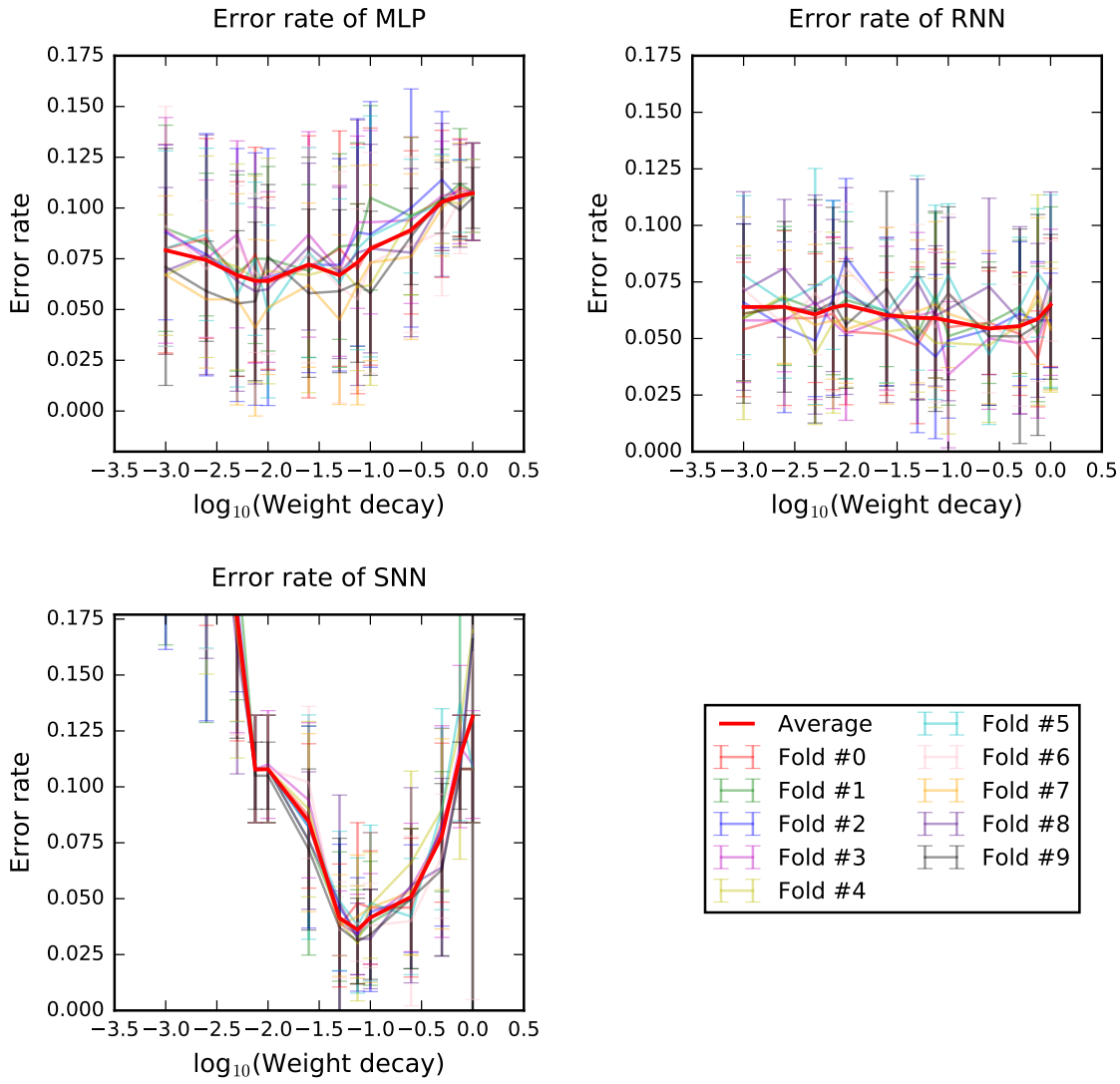


Figure 4.6: Overall trend of the error rate with varying weight decay.

4.4 Evaluation Metrics

Due to the imbalance in epigenetic datasets, the accuracy is not sufficient to assess the performance of the models, since a model always predicting the most-represented class can achieve a reasonably high accuracy. Additional evaluation metrics were chosen to obtain a better indication of the discriminative ability of the models.

4.4.1 Binary Classification

For a given binary classifier, the performance metrics are computed using the number of true positives (TP), false positives (FP), true negatives (TN) and false negatives

(FN). These values can be obtained by using the confusion matrix of the binary classifier, described in Figure 4.7.

		Predicted Class: Positive Class	Predicted Class: Negative Class
Actual Class	Positive Class	True Positives (TP)	False Negatives (FN)
	Negative Class	False Positives (FP)	True Negatives (TN)

Figure 4.7: Confusion matrix of a binary classifier

The metrics used to evaluate the NNs are:

- **Accuracy**:
$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.1}$$
- **Precision**:
$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.2}$$
- **Sensitivity**: (same as recall) - true positive rate
$$\text{Sensitivity} = \frac{TP}{TP + FN} \tag{4.3}$$
- **F1-score**: harmonic mean of precision and sensitivity
$$F1 = \frac{2TP}{2TP + FP + FN} \tag{4.4}$$
- **Matthews Correlation Coefficient (MCC)**
$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{4.5}$$

The accuracy, precision, sensitivity and $F1$-score have values in the range $[0, 1]$, with values closer to 1 indicating better performance. The MCC has values in the range $[-1, 1]$ with 1 indicating perfect prediction, 0 no better than random and -1 representing total disagreement between the true label and prediction.

Amongst these metrics, MCC gives a balanced measure for evaluating a binary classifier, especially for imbalanced data. The other metrics, are more biased towards identifying the elements of the positive class.

A Receiver Operating Characteristic (ROC) curve plots the true positive rate against the false positive rate at different threshold values. The area under the ROC curve (the AUC) measures the probability that a positive example will receive a higher confidence in its positivity than a negative example[1].

[1]Arguably, for binary classification the AUC should actually be as far as possible from 0.5,

4.4.2 Multiclass Classification

The confusion matrix can easily be extended to multiple classes. Assuming a One-vs-All classifier, each class can be, in turn, considered as the "positive class", while the other classes form the "negative class". For each class i the metrics TP_i, FP_i, TN_i and FN_i are derived from the confusion matrix.

By considering Class 2 to be the "positive class" in the confusion matrix represented in Figure 4.8, TP_2, FP_2, TP_2, TN_2 can be derived by summing over the cells that contain their name.

		Predicted Class			
		Class 0	Class 1	Class 2	Class 3
Actual Class	Class 0	TN	TN	FP	TN
	Class 1	TN	TN	FP	TN
	Class 2	FN	FN	TP	FN
	Class 3	TN	TN	FP	TN

Figure 4.8: Confusion matrix for multiple classes.

The evaluation metrics computed for binary classification can be extended to multiple classes by using micro-averaging and macro-averaging. Micro-averaging assigns equal weights to each test example; macro-averaging gives the same weight to each class. Consequently, the micro-averaged value can be dominated by the classes with most examples.

Assume that there are k classes, and let $B\Big(TP, FP, TN, FN\Big)$ be an evaluation metric used for binary classification. The micro- and macro- extensions of B to multiple classes are computed as:

$$B_{micro} = B\left(\sum_{i=1}^{k} TP_i, \sum_{i=1}^{k} FP_i, \sum_{i=1}^{k} TN_i, \sum_{i=1}^{k} FN_i\right)$$

$$B_{macro} = \frac{1}{k}\sum_{i=1}^{k} B\Big(TP_i, FP_i, TN_i, FN_i\Big)$$

The evaluation metrics that are most relevant for multi-class classification are the F1-score, MCC and accuracy. The accuracy will be computed as the ratio between correctly classified examples and the total number of examples, rather than through micro- or macro-averaging.

4.4.3 Paired t-test

The use of k-fold cross-validation also integrates well with paired t-tests for assessing statistical significance of the observed differences between two models with respect to a performance metric.

because if it is near-zero, one may simply invert the predictions of the classifier to obtain good performance.

This test seeks to potentially reject the null hypothesis, which states that the two models have the same fundamental performance, and that any differences observed between them are due to random noise. By using the same order of k folds in the outer CV the performances of two of the NNs can be paired. The p-value is computed for the two-tailed Student t-distribution and the threshold value used for rejecting the null hypothesis is $p < 0.05$.

4.5 Evaluate models on cancer patients dataset

The dataset for the cancer patients contains 590 training examples, out of which 530 are positive (the patient has cancer) and 60 are negative. The evaluation metrics are computed over 10-fold outer CV.

The input data for each example consists of DNA methylation levels and gene expression levels. The SNN employs both types of data, such that each modality is given as input to a different superlayer. Conversely, the MLP and RNN are assessed based on their predictions when the two types of data are concatenated, but also when a single modality is employed.

4.5.1 Cancer patients data with DNA methylation and gene expression levels

The MLP and RNN were firstly used to make predictions based on the two modalities concatenated, while the SNN used each modality as input to a different superlayer. The results are summarised in Tables 4.1 and 4.2.

Metric	**MLP**	**RNN**	**SNN**
Accuracy	0.934 ± 0.039	0.950 ± 0.021	$\mathbf{0.981} \pm 0.016$
Precision	0.948 ± 0.029	0.965 ± 0.021	$\mathbf{0.996} \pm 0.007$
Sensitivity	0.980 ± 0.016	0.981 ± 0.017	$\mathbf{0.983} \pm 0.018$
F1 - score	0.964 ± 0.021	0.973 ± 0.011	$\mathbf{0.989} \pm 0.009$
MCC	0.606 ± 0.247	0.723 ± 0.126	$\mathbf{0.912} \pm 0.072$

Table 4.1: Results obtained for each model after 10-fold outer CV.

Metric	MLP/RNN	MLP/SNN	RNN/SNN
Accuracy	$p = 0.38448$	$p = 0.00367$	$p = 0.01520$
Precision	$p = 0.32176$	$p = 0.00064$	$p = 0.00272$
Sensitivity	$p = 0.952412$	$0 = 0.80883$	$p = 0.85333$
F1 - score	$p = 0.39325$	$p = 0.00367$	$p = 0.01689$
MCC	$p = 0.33851$	$p = 0.00227$	$p = 0.00802$

Table 4.2: p-values obtained for pair-wise comparison of the models.

By obtaining higher metric averages, the SNN outperforms the MLP and RNN. The comparison between the SNN and MLP, and the SNN and RNN is also statistically significant for certain performance metrics. These results were expected, since the SNN most explicitly exploits the interactions between the different modalities present in the dataset.

While the RNN achieves higher metric averages than the MLP, the p-values indicate that the models perform similarly. More insight into why this may be the case is given in Figure 4.9, where the MCC is illustrated across different folds. The plot shows that for some folds the MLP gives better results, RNN prevails in others. Nevertheless, on the majority of folds, the SNN displays the highest MCC.

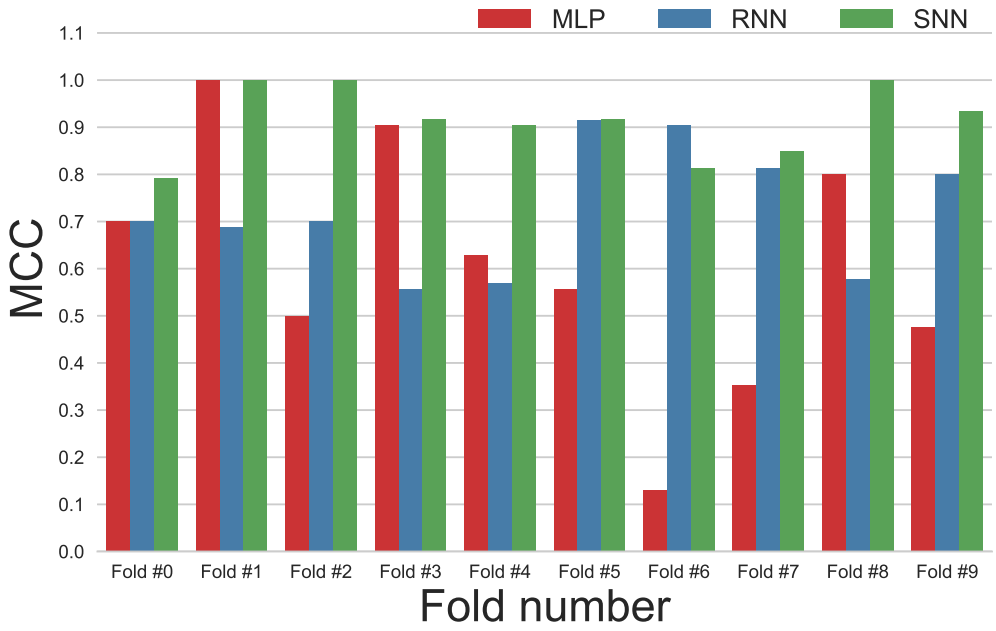


Figure 4.9: Per-fold MCC value for each model.

Figure 4.10 illustrates the mean ROC curve for each model, obtained by interpo-

lating the ROC curves computed after each fold. The ROC curve for the SNN is the steepest, indicating that the SNN has the highest ability to discriminate between the two classes. Concavities present in the curves of the MLP and RNN indicate that locally, these models have worse than random behaviour.

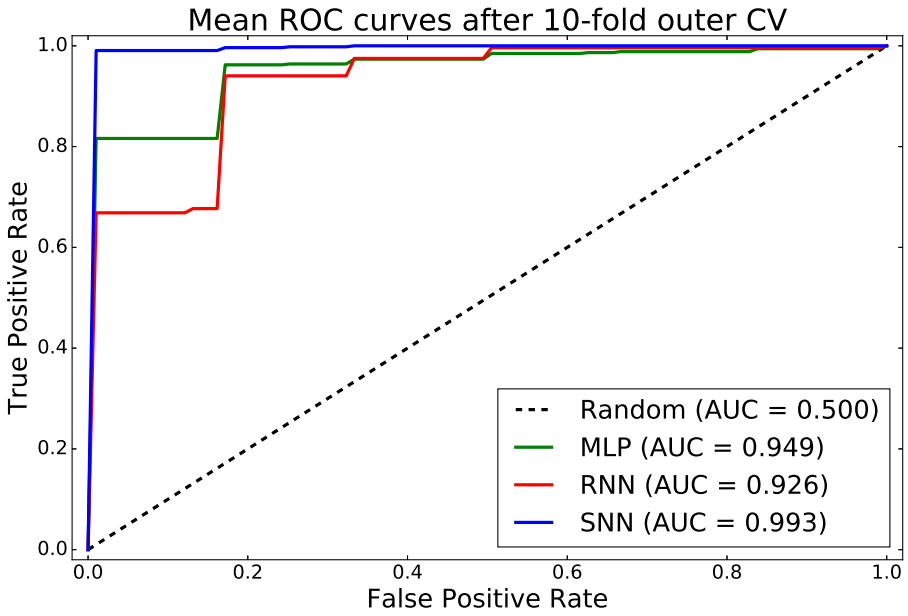


Figure 4.10: Mean ROC curves obtained after 10-fold outer CV.

4.5.2 Cancer patients data with DNA methylation levels

The lower performance of the MLP and RNN compared to the SNN may be attributed to the fact that, under the small number of samples provided, the former models were not able to capture the relationships between the two different modalities. Hence, I decided to compare the MLP and RNN in the situation when they are only given as input DNA Methylation levels. This represents a crude form of dimensionality reduction. The results are summarised in Table 4.3.

Metric	**MLP**	**RNN**	**SNN**
Accuracy	0.969 ± 0.018	0.974 ± 0.023	$\mathbf{0.981} \pm 0.016$
Precision	0.994 ± 0.008	0.992 ± 0.009	$\mathbf{0.996} \pm 0.007$
Sensitivity	0.971 ± 0.021	0.978 ± 0.025	$\mathbf{0.989} \pm 0.018$
F1 - score	0.982 ± 0.011	0.985 ± 0.013	$\mathbf{0.989} \pm 0.009$
MCC	0.863 ± 0.078	0.881 ± 0.101	$\mathbf{0.912} \pm 0.071$

Table 4.3: Results obtained for each model after 10-fold outer CV.

Due to the similar performance of the models, the p-values were omitted and the MCC values were plotted instead in Figure 4.11.

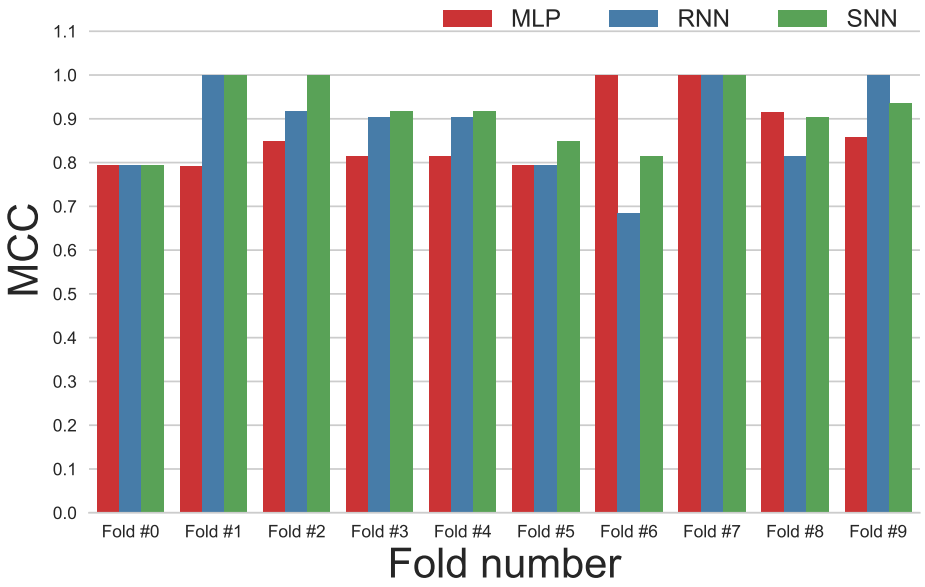


Figure 4.11: Per-fold MCC value for each model.

As further illustrated by the mean ROC curves plotted in 4.12, the performance of the MLP and RNN significantly increases and is now comparable to the performance of the SNN. Nevertheless, the SNN still achieves the highest average results and MCC values in all folds (but one), further indicating that the SNN is better at capturing relationships at lower data levels. Consequently, while there is insufficient data to make a statistically sound conclusion, the odds are clearly in SNN's favour.

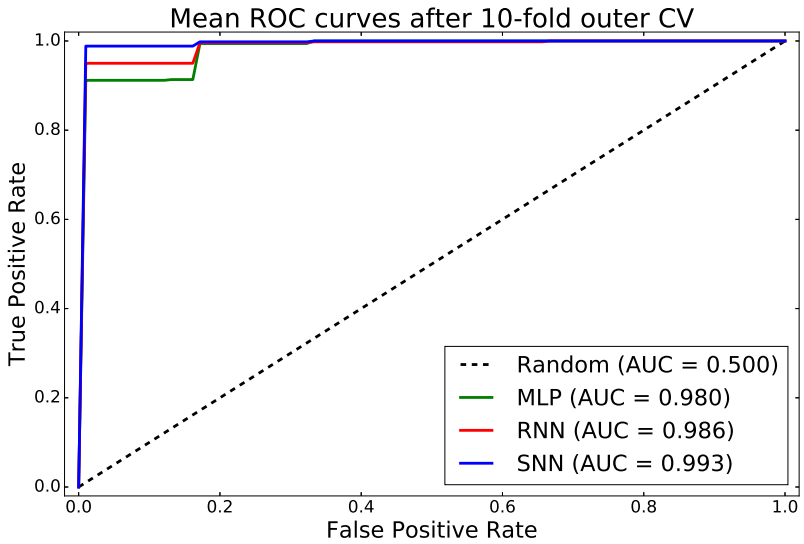


Figure 4.12: Mean ROC curves obtained after 10-fold outer CV.

4.6 Evaluate models on Embryo Development data

The dataset contains 90 training examples which are unevenly distributed over 7 classes, representing the following embryonic development stages: *Oocyte, Zygote, 2-cell embryo, 4-cell embryo, 8-cell embryo, Morulae, Late blastocyst.* The input data for the MLP and RNN was obtained by selecting 128 genes that have the highest entropy across the different classes, while the k-means clustering algorithm was used to form the input to each superlayer of the SNN.

The results are summarised in Table 4.4. Due to the limited number of available examples and high-variability in the evaluation metrics caused by a single misclassified example, the p-values obtained were too high to ascertain statistical significance for rejecting the null hypothesis. Consequently, the accuracy of the models for each fold was plotted in Figure 4.13 to gain more insight into their performance.

Metric	MLP	RNN	SNN
Micro F1-score	0.934 ± 0.067	0.895 ± 0.062	$\mathbf{0.976} \pm 0.034$
Micro MCC	0.923 ± 0.078	0.878 ± 0.007	$\mathbf{0.972} \pm 0.039$
Macro F1-score	0.854 ± 0.144	0.902 ± 0.063	$\mathbf{0.958} \pm 0.070$
Macro MCC	0.856 ± 0.145	0.862 ± 0.078	$\mathbf{0.956} \pm 0.070$
Accuracy	0.931 ± 0.067	0.890 ± 0.063	$\mathbf{0.975} \pm 0.035$

Table 4.4: Results obtained for each model after 6-fold outer CV.

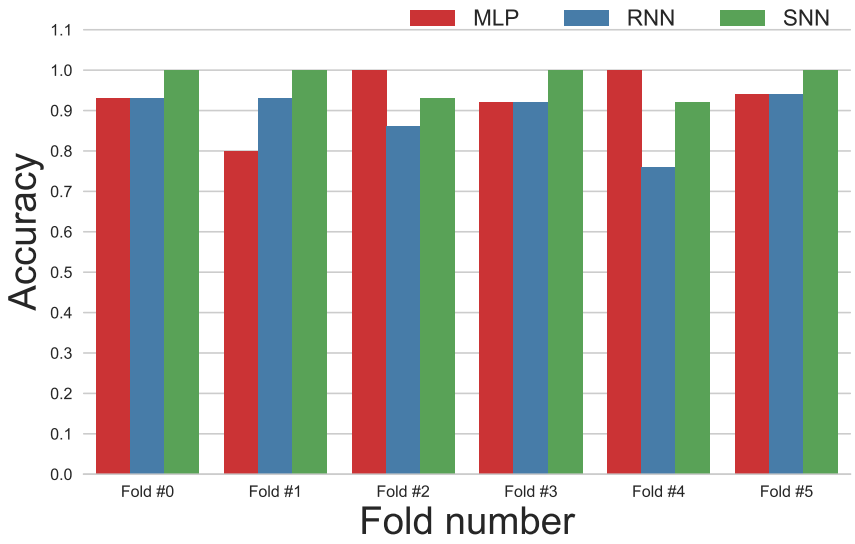


Figure 4.13: Per-fold accuracy of each model. The SNN achieves perfect classification of 4 out of the 6 folds.

It is noticeable that the MLP has higher micro-averaged scores, while the RNN has higher macro-averaged scores. The micro-averaged score gives equal importance to each test example, which means that the MLP correctly classified more of the test examples. Conversely, the macro-averaged score gives equal importance to each class. Since the dataset is highly unbalanced, these results highlight that the RNN can distinguish between the examples from under-represented classes better, as emphasised by the confusion matrices in Figures 4.14 and 4.15.

By properly exploiting the relationships between the input clusters, the SNN achieves the highest average evaluation metrics. Nevertheless, the confusion matrix (Figure 4.16) depicts that the SNN also mis-classified an example from the under-represented Zygote class, on which the RNN achieves perfect classification.

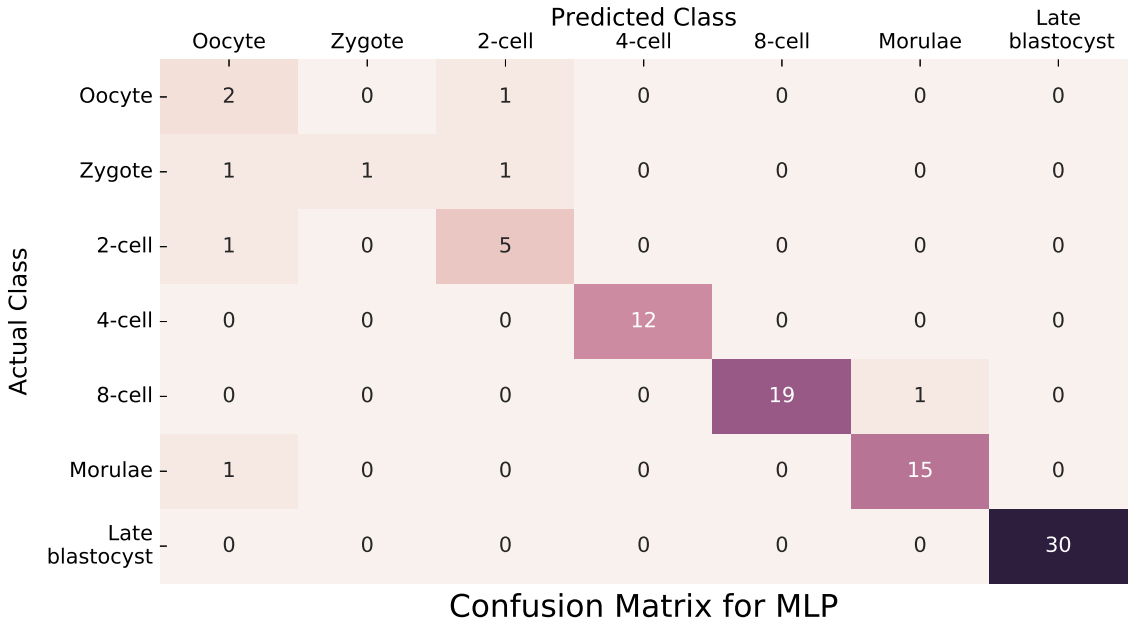


Figure 4.14: The MLP shows poor performance on examples from under-represented classes, but achieves almost perfect performance on the overly-represented class.

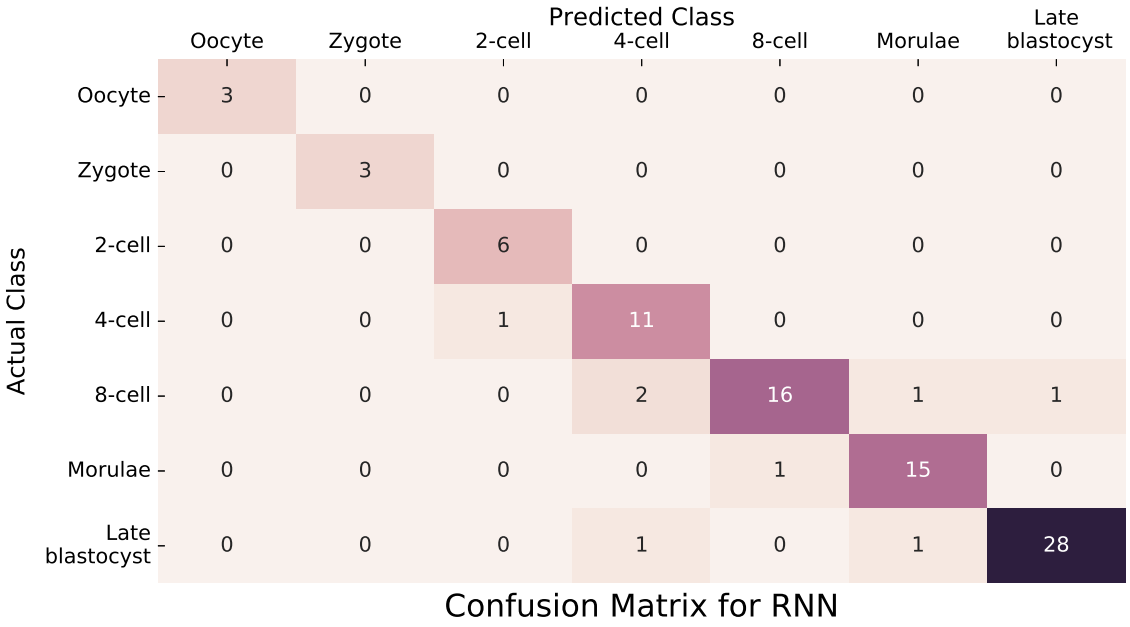


Figure 4.15: The RNN perfectly classifies examples from the under-represented classes, namely the Oocyte, Zygote and 2-cell embryo.

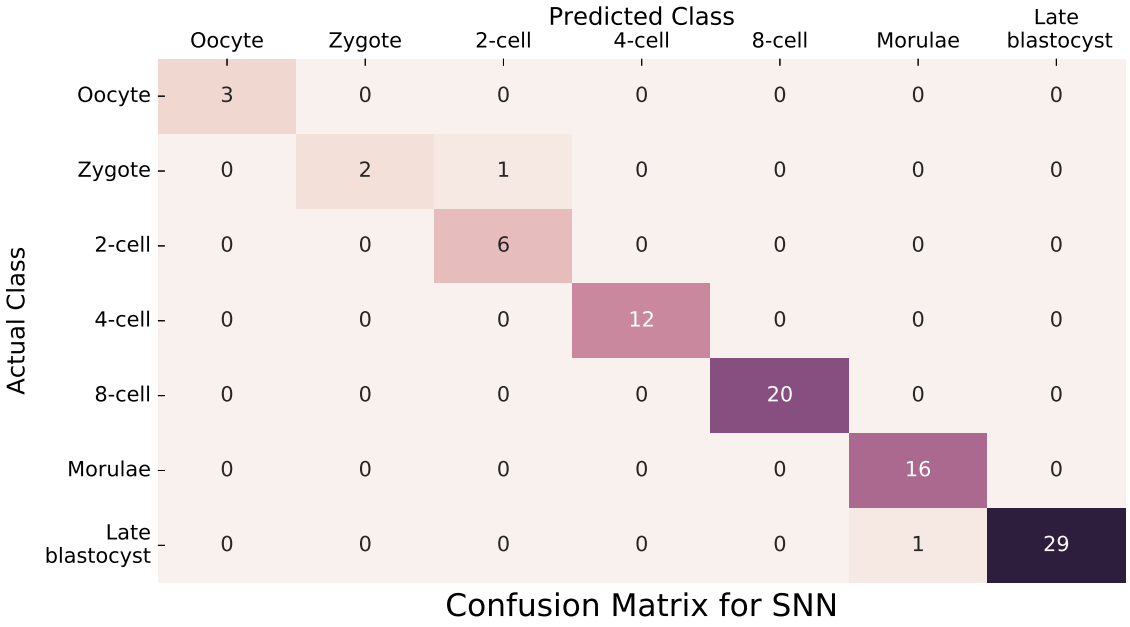


Figure 4.16: The SNN achieves almost perfect classification for examples across all classes; only 2 out of 90 examples misclassified over 6-fold outer CV.

4.6.1 Evaluate Clustering Algorithms

The different approaches taken by the two clustering algorithms implemented require that their impact on the SNN are evaluated. Additionally, it was appropriate to use

data from a single cluster as input to the RNN and MLP to determine whether the performance of these models increases. The clustering algorithms are also compared in this situation. The results are presented in Tables 4.5, 4.6, 4.7.

Metric	MLP/k-means clustering data	MLP/hierarchical clustering data	p-**value**
Micro F1-score	**0.929** $\pm$ 0.038	0.907 $\pm$ 0.076	0.60396
Micro MCC	**0.918** $\pm$ 0.045	0.891 $\pm$ 0.089	0.60396
Macro F1-score	**0.867** $\pm$ 0.088	0.791 $\pm$ 0.159	0.29911
Macro MCC	**0.865** $\pm$ 0.085	0.787 $\pm$ 0.163	0.31813
Accuracy	**0.923** $\pm$ 0.041	0.905 $\pm$ 0.077	0.62654

Table 4.5: Average results after 6-fold outer CV on the performance of the MLP on classifying input data obtained from k-means clustering and hierarchical clustering.

Metric	RNN/k-means clustering data	RNN/Hierarchical clustering data	p-**value**
Micro F1-score	**0.805** $\pm$ 0.072	0.552 $\pm$ 0.129	0.00170
Micro MCC	**0.773** $\pm$ 0.084	0.478 $\pm$ 0.151	0.00170
Macro F1-score	**0.712** $\pm$ 0.112	0.420 $\pm$ 0.135	0.02782
Macro MCC	**0.687** $\pm$ 0.118	0.356 $\pm$ 0.155	0.02780
Accuracy	**0.801** $\pm$ 0.071	0.548 $\pm$ 0.131	0.00186

Table 4.6: Average results after 6-fold outer CV on the performance of the RNN on classifying input data obtained from k-means clustering and hierarchical clustering.

Metric	SNN/k-means clustering data	SNN/Hierarchical Clustering data	p**-value**
Micro F1-score	$\mathbf{0.964} \pm 0.035$	0.958 ± 0.043	0.78702
Micro MCC	$\mathbf{0.956} \pm 0.041$	0.952 ± 0.050	0.78702
Macro F1-score	$\mathbf{0.951} \pm 0.061$	0.942 ± 0.069	0.62841
Macro MCC	$\mathbf{0.950} \pm 0.062$	0.939 ± 0.069	0.59980
Accuracy	$\mathbf{0.963} \pm 0.036$	0.956 ± 0.045	0.78904

Table 4.7: Average results after 6-fold outer CV on the performance of the SNN on classifying input data obtained from k-means clustering and hierarchical clustering.

The results highlight that using k-means clustering to pre-process the input to the NNs gives better results over hierarchical clustering. Nevertheless, by comparing these results with those obtained previously, it is noticeable that performance of the MLP and RNN is not improved by using input data from a cluster, which means that the models are not aided by the increased correlation in the input data.

4.7 Summary

The chapter initially described how the models have successfully achieved their expected performance on synthetic data. Comparative evaluation has shown that the SNN outperforms the MLP and RNN by exploring different modalities in the epigenetic data. The similar performance of the MLP and RNN indicates that it makes little difference whether epigenetic data input is processed all at once or sequentially. Moreover, the comparison between the clustering algorithms shows that k-means produces better pre-processing of epigenetic data.

Chapter 5

Conclusions

This dissertation has outlined the work done in designing, implementing and evaluating three different types of NNs for performing inference on epigenetic data. The problem of having sparse and imbalanced epigenetic datasets was shown to be overcome if suitable pre-processing techniques and NN architectures are employed.

5.1 Results

The project requirements were successfully achieved and surpassed by implementing the additional extensions. The results obtained highlight that the use of NN models to study epigenetic datasets has been highly successful as per the metrics defined in the Evaluation chapter.

The use of NNs to make predictions based on epigenetic data is highly applicable in medical diagnosis, where NNs can be used to identify patterns in epigenetic modifications that are specific to the growth of tumours or in ensuring the healthy development of embryos.

The project has been an enjoyable experience and has helped me enhance my knowledge in designing neural network architectures. Since my future interests involve working on research in machine learning applications to biomedical datasets, the skills acquired while developing this project will be invaluable.

5.2 Lessons learnt

This project has given me the opportunity to explore different NN models to find the most appropriate architecture for epigenetic data. Since the project did not follow well-established architectures published in literature, significant time was put into deciding on the most appropriate design for each type of NN.

Using the iterative development model for this process proved to be crucial. If I were to do the project again, I would have allowed for even more time to try out different techniques to improve the NN models. This is particularly important for machine learning projects, where there usually is not a pre-defined sequence of steps

that need to be followed to obtain good results, but rather, there is a lot of trial and error involved.

The choice of Python was also appropriate and its support for TensorFlow and ease of data handling allowed for smooth development of the project.

5.3 Further Work

The comparative analysis of the NNs has shed light on avenues that can be explored, including:

- *Using more advanced pre-processing techniques:* An example of this includes work done by Romero *et al.* [25] who employed an MLP to learn a distributed representation of each input feature and then used this representation as input to another MLP performing the classification task.
- *Extending the SNN to incorporate more superlayers:.* This can be particularly beneficial because it will allow the introduction of additional data clusters or modalities in the pattern recognition process.
- *Exploring a superlayered architecture with superlayers consisting of RNNs*: since the SNN gave the best results on epigenetic data, it comes as a natural extension to try to a different types of architecture for each superlayer.

Final Remarks

With the foundations set by this project and development of ideas to be explored in future work, I hope that architectures established in the project can be exploited to not only further medical research but eventually form an integral part of systems deployed in healthcare institutions for routine diagnosis.

Bibliography

[1] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine learning*, 75(2):245–248, 2009.

[2] Christof Angermueller, Heather Lee, Wolf Reik, and Oliver Stegle. Accurate prediction of single-cell dna methylation states using deep learning. *bioRxiv*, page 055715, 2016.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[4] Pierre Baldi, Søren Brunak, Paolo Frasconi, Giovanni Soda, and Gianluca Pollastri. Exploiting the past and the future in protein secondary structure prediction. *Bioinformatics*, 15(11):937–946, 1999.

[5] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[6] Ernesto Estrada and Jesús Gómez-Gardeñes. Communicability reveals a transition to coordinated behavior in multiplex networks. *Physical Review E*, 89(4):042819, 2014.

[7] Andrew P Feinberg, Michael A Koldobskiy, and Anita Göndör. Epigenetic modulators, modifiers and mediators in cancer aetiology and progression. *Nature Reviews Genetics*, 2016.

[8] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[9] Clara Granell, Sergio Gómez, and Alex Arenas. Competing spreading processes on multiplex networks: awareness and epidemics. *Physical review E*, 90(1):012808, 2014.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[11] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N

Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[14] Oliver Kaut, Ina Schmitt, Jörg Tost, Florence Busato, Yi Liu, Per Hofmann, Stephanie H Witt, Marcella Rietschel, Holger Fröhlich, and Ullrich Wüllner. Epigenome-wide dna methylation analysis in siblings and monozygotic twins discordant for sporadic parkinson's disease revealed different epigenetic patterns in peripheral blood mononuclear cells. *neurogenetics*, pages 1–16, 2016.

[15] Mikko Kivelä, Alex Arenas, Marc Barthelemy, James P Gleeson, Yamir Moreno, and Mason A Porter. Multilayer networks. *Journal of complex networks*, 2(3):203–271, 2014.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[17] Craig Larman and Victor R Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, 2003.

[18] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

[19] Byunghan Lee, Junghwan Baek, Seunghyun Park, and Sungroh Yoon. deeptarget: end-to-end learning framework for microrna target prediction using deep recurrent neural networks. *arXiv preprint arXiv:1603.09123*, 2016.

[20] Byunghan Lee, Taehoon Lee, Byunggook Na, and Sungroh Yoon. Dna-level splice junction prediction using deep recurrent neural networks. *arXiv preprint arXiv:1512.05135*, 2015.

[21] Polina Mamoshina, Armando Vieira, Evgeny Putin, and Alex Zhavoronkov. Applications of deep learning in biomedicine. *Molecular Pharmaceutics*, 2016.

[22] Seonwoo Min, Byunghan Lee, and Sungroh Yoon. Deep learning in bioinformatics. *Briefings in bioinformatics*, 2016.

[23] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[24] Seunghyun Park, Seonwoo Min, Hyunsoo Choi, and Sungroh Yoon. deepmirgene: deep neural network based precursor microrna prediction. *arXiv preprint arXiv:1605.00017*, 2016.

[25] Adriana Romero, Pierre Luc Carrier, Akram Erraqabi, Tristan Sylvain, Alex Auvolat, Etienne Dejoie, Marc-André Legault, Marie-Pierre Dubé, Julie G Hussin, and Yoshua Bengio. Diet networks: Thin parameters for fat genomic. *arXiv preprint arXiv:1611.09340*, 2016.

[26] Jose V Sanchez-Mut and Johannes Gräff. Epigenetic alterations in alzheimer's disease. *Frontiers in behavioral neuroscience*, 9, 2015.

[27] Ritambhara Singh, Jack Lanchantin, Gabriel Robins, and Yanjun Qi. Deepchrome: deep-learning for predicting gene expression from histone modifications. *Bioinformatics*, 32(17):i639–i648, 2016.

[28] Søren Kaae Sønderby and Ole Winther. Protein secondary structure prediction with long short term memory networks. *arXiv preprint arXiv:1412.7828*, 2014.

[29] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[30] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[31] Ioannis Valavanis, Eleftherios Pilalis, Panagiotis Georgiadis, Soterios Kyrtopoulos, and Aristotelis Chatziioannou. Cancer biomarkers from genome-scale dna methylation: Comparison of evolutionary and semantic analysis methods. *Microarrays*, 4(4):647–670, 2015.

[32] Tim Van den Bulcke, Koenraad Van Leemput, Bart Naudts, Piet van Remortel, Hongwu Ma, Alain Verschoren, Bart De Moor, and Kathleen Marchal. Syntren: a generator of synthetic gene expression data for design and analysis of structure learning algorithms. *BMC bioinformatics*, 7(1):43, 2006.

[33] Petar Veličković, Duo Wang, Nicholas D. Lane, and Pietro Liò. X-cnn: Cross-modal convolutional neural networks for sparse datasets. *IEEE Symposium Series on Computational Intelligence*, 2016.

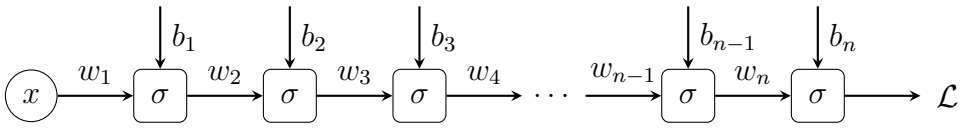
[34] Athina Vidaki, David Ballard, Anastasia Aliferi, Thomas H Miller, Leon P Barron, and Denise Syndercombe Court. Dna methylation-based forensic age prediction using artificial neural networks and next generation sequencing. *Forensic Science International: Genetics*, 2017.

[35] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

Appendix A

Vanishing Gradients

Consider an NN architecture that has the form of a path where each layer consists of a single neuron, and let $\mathcal{L}$ be the loss of this network computed during a training iteration.



Let z_i be the input to the neuron in the i-th layer and let a_i be the activation of the neuron in the same layer. The expressions for z_i and a_i can be computed as follows:

$$z_i = w_i a_{i-1} + b_i \tag{A.1}$$

$$a_i = \sigma(z_i) \tag{A.2}$$

While training this network, the derivative of the loss function $\mathcal{L}$ with respect to the weights and biases needs to be evaluated. This is achieved by using the chain rule during backpropagation. Therefore, the partial derivative of the loss function with respect to w_1 is given by:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} \tag{A.3}$$

$$= x \cdot \sigma'(z_1) \cdot \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \tag{A.4}$$

$$= x \cdot \sigma'(z_1) \cdot w_2 \cdot \sigma'(z_2) \cdot \frac{\partial \mathcal{L}}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial a_2} \tag{A.5}$$

$$= \ldots \tag{A.6}$$

$$= x \cdot \sigma'(z_1) \cdot w_2 \cdot \sigma'(z_2) \cdot w_3 \cdot \sigma'(z_3) \cdot \cdots \cdot \frac{\partial \mathcal{L}}{\partial a_n} \tag{A.7}$$

Neural network models have many such paths, over which the gradient updates are multiplied together. In particular, in an RNN, the length of such a path is at least the number of time-steps in the input sequence.

It is important to notice that the gradient of the loss function with respect to the weight in the first layer is computed as the product of the weights and gradients in all the later layers.

Therefore, when $|\sigma'(z)| \leq 1$, which is the case for the logistic and tanh activation function the gradients in the network will get very small. This problem is known as **vanishing gradients**. As a consequence, the neurons in the first layers of the network will learn much more slowly than the neurons in later layers.

Appendix B

Adam Optimiser

Given the gradient of loss function with respect to the weights over the mini-batch $\mathcal{B}$ at timestep t:

$$\mathbf{g}_t = \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{B}} \frac{\partial \mathcal{L}(\mathbf{w}, \mathbf{b}, \mathbf{x}, \mathbf{y})}{\partial \mathbf{w}},$$

Adam Optimiser computes an exponentially decaying average of previous gradients $\mathbf{m}_t$ and an exponentially decaying average of previous squared gradients $\mathbf{v}_t$.

$$\mathbf{m}_t = \beta_1 \cdot \mathbf{m}_{t-1} + (1 - \beta_1) \cdot \mathbf{g}_t \tag{B.1}$$

$$\mathbf{v}_t = \beta_2 \cdot \mathbf{v}_{t-1} + (1 - \beta_2) \cdot \mathbf{g}_t^2 \tag{B.2}$$

where β_1 and β_2 are the exponential decaying rates and are usually set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

$\mathbf{m}_t$ and $\mathbf{v}_t$ are biased since they are initialised to zero. Therefore, the following correction is applied:

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m_t}}{1 - \beta_1^t} \tag{B.3}$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_2^t} \tag{B.4}$$

Appendix C

Robustness to noise

Experiments for extracting epigenetic data are susceptible to errors. Thereupon, it is essential to assert that the performance of the NNs is not severely affected by this noise introduced into the data due to experimental errors. Hence, the class for the epigenetic data was extended to allow for the introduction of Gaussian noise $\mathcal{N}(0, \sigma^2)$ into the datasets.

The robustness of the models to noise was tested on the data for the cancer patients, for which 10-fold datasets were obtained for values of σ^2 in the range (0, 2.5). The average accuracy and MCC values were plotted in Figures C.1 and C.2.

The model that is most affected by noise is the SNN. In addition, when the noise variance is within the range $(0.1, 0.7)$ the RNN surpasses the performance of the MLP. However, with increased noise variance the MLP gives the best results.

Nevertheless, the results show that the NNs have good robustness to noise.

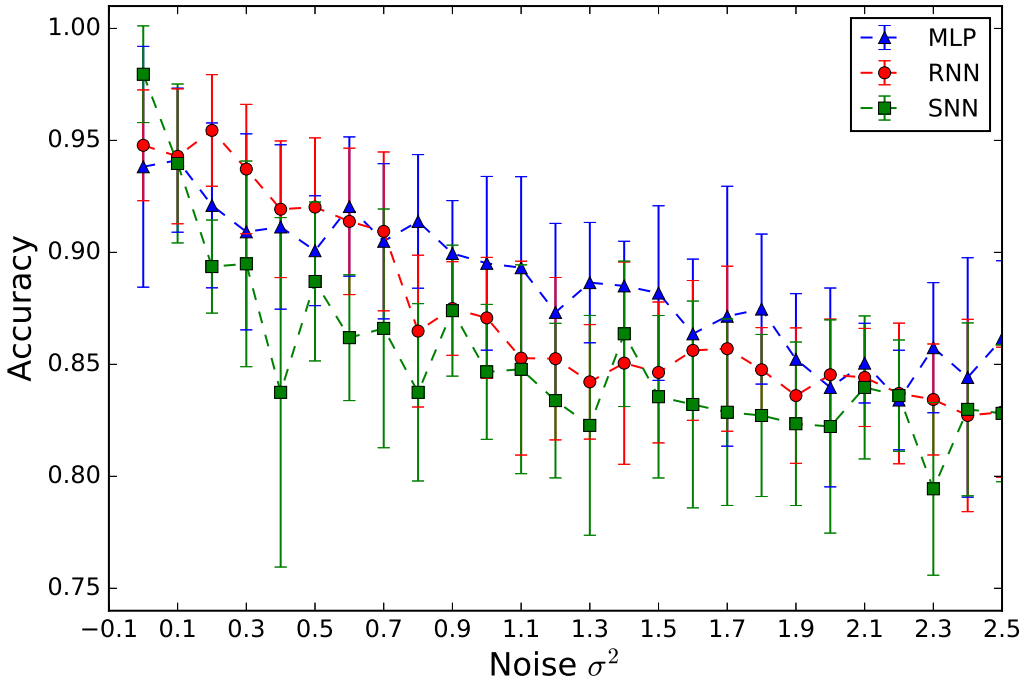


Figure C.1: Average accuracy after 10-fold outer CV plotted for each model against the different variances of the noise introduced in each dataset.

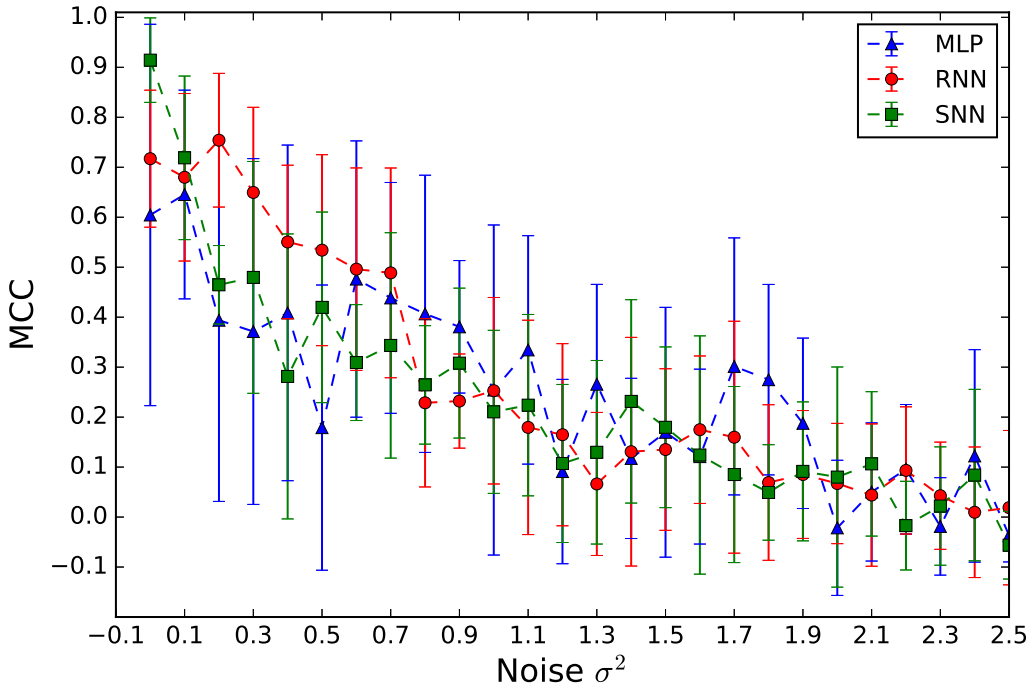


Figure C.2: Average MCC value after 10-fold outer CV plotted for each model against the different variances of the noise introduced in each dataset.

Appendix D

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Comparative Analysis of Neural Network Architectures for Epigenetics Inference

Ioana Bica, Churchill College
Originator: Petar Veličković
May 19, 2017

Project Supervisor: Dr Pietro Lio'
Director of Studies: Dr John Fawcett
Project Overseers: Prof. Jean Bacon & Prof. Ross Anderson

D.1 Introduction and Description of Work

D.1.1 Description of the problem

Epigenetics represents the study of how variations in gene expressions cause modifications of organisms. Specifically, epigenetics analyses changes in the phenotype of the cells without changes in the underlying DNA sequence of the cells. Though epigenetic variations are normal, there are many factors that can be induced by age, lifestyle, environment as well as diseases. By performing inference tasks on epigenetic data, one can learn extensively about how cells change during the first stages of an organism's life, how stem cells differentiate into more specialised cells—brain, heart, skin or a range of other cells types—and perhaps most importantly, how epigenetic modifications cause diseases like cancer later on in life.

Epigenetic data is high dimensional and complex, which makes the data difficult for humans to interpret. Nonetheless, machine learning techniques can facilitate the analysis of epigenetic data which can bring great benefits to biomedical research. [21, 22]

Machine learning methods based on neural networks perform very well on complex and high dimensional data which makes them suitable for studying epigenetics. In addition, deep neural networks are capable of extracting the non-linear relationships in the data and are also able to generalise well on new data.

However, the main problem that arises with epigenetic and, in general, medical data, is that the datasets are sparse and imbalanced. The deep learning models are very complex and, as a consequence, they usually require large training datasets to be able to optimise their parameters.

In addition, having imbalanced data is problematic, particularly in classification where the model may become biased towards the overrepresented classes.

My project will address the problem of having sparse and imbalanced epigenetic datasets for training deep learning models by comparing neural network architectures that take advantage instead of the width and sequential structure of the data.

D.1.2 Project Outline

Although large volumes of training examples are rare in epigenetics, a single sample can provide substantial information. The datasets usually have thousands of gene expressions within one training sample. This represents a very useful feature of the data that can be exploited in the neural network architectures.

The classical feedforward neural networks architecture can be modified to account for the sparsity and structure of the epigenetic data.

The epigenetics dataset consists of a matrix where the rows are identified by genes and the columns are identified by the biological cells. A value in the matrix represents the expression level of the gene for the particular cell. A training example consists of one column in the matrix and this will be the input to the neural network architectures.

My project will involve implementing and comparing several neural network architectures which exhibit specific characteristics that make them appropriate for analysing epigenetic data.

The main architectures that will be compared are:

- multilayer perceptron neural networks optimised for data sparsity
- recurrent neural networks
- superlayered feedforward neural networks

The reason for choosing to use the recurrent neural network is because it takes advantage of the sequential form of the epigenetic data, while the superlayered networks architecture explores the heterogeneity of this data.

Furthermore, clustering of genes in the datasets can also yield more targeted exploitation of data sparsity. Therefore, a clustering algorithm will also be implemented and each cluster will be used as a layer in the superlayered networks.

These techniques will be validated by using synthetically created datasets consisting of specific gene patterns or on data found in the literature that already includes experimentally determined patterns. The artificially created datasets allow

us to fully control the statistical properties of the data to be tested. [32] As a result, the output of the neural network can be tested against the expected accuracy on the synthetic data.

The evaluation of the neural network architectures will take into consideration the performance of the neural networks on the prediction tasks performed on the epigenetics data. Assessment metrics will also be used to clearly observe how the performance of the architectures is affected by the sparsity and imbalance in the data. The F-measure, for instance, which is the harmonic mean of precision and recall can provide a more insightful performance measure than simply computing the accuracy. Moreover, the area under the receiver operating characteristic curve (AUC) and the area under the precision recall curve (AUC-PR) can be used as metrics for measuring the performance over different class distribution.

D.2 Starting point

The project will be based on the following prior knowledge and existing libraries:

- **Computer Science Tripos Course**
  - ***Artificial Intelligence I (Part IB)***: the course provides an introduction to machine learning; in addition, the course covers the basic structure of a feedforward neural netowork and the ways in which it can be trained using gradient descent and backpropagation
  - ***Bioinformatics (Part II)***: since the project is based on biomedical data, the course will provide in depth knowledge about the structure of the data and about the algorithms used for analysing this type of data
  - ***Machine Learning and Bayesian Inference (Part II)***: the course dives deeper into machine learning models, however, since it is a Lent term course I will have to study it in advance
  - ***Software Engineering (Part IB)***: the course has equipped me with the necessary practices for developing complex projects.
- **Open Source Libraries**
  - ***TensorFlow***: open source library developed by Google for performing numerical computation using dataflow graphs. TensorFlow provides support for specifying the hyperparameters in the neural network architectures such as the structure of the layers, number of hidden layers and the connections between the layers. My project will be developed in Python and will use the Python API of TensorFlow for building the neural network architectures.
- **Programming experience**
  - ***Python***: I have experience programming in Python from the group project in Part IB.

D.3 Substance and Structure of the Project

The work for the project will be divided in the modules presented below. However, these modules will not be strictly independent. The preparation part will take place before any major architecture implementation and the dissertation write-up will also happen concurrently with the preparation, implementation and evaluation of the architectures.

D.3.1 Preparation

As mentioned above, the preparation will be divided in multiple stages, which will be interspersed between the other modules of my project. During the preparation stages I will familiarise myself with the models that need to be implemented and with TensorFlow. This will be done by reading books and academic papers to understand how each neural network architecture learns to perform the inference tasks. Additional papers on using deep learning in bioinformatics will be studied in order to understand which are the best practices for choosing the hyperparameters of the networks for epigenetic data. In addition, I am also planning to do an online course on deep learning and tutorials to learn how to use TensorFlow.

Before implementing the superlayered architecture I will also read several papers about clustering algorithms used on epigenetic data in order to understand how each one of them exploits the underlying structure of this type of data.

The aims of the preparation stages are the following:

- fully understand how feedforward neural networks, recurrent neural networks and superlayered networks work and how each one of them learns from the data
- get familiar with clustering algorithms and fully understand how the clustering algorithm chosen for implementation works
- get familiar with TensorFlow for the implementation of the neural network architectures

The relevant knowledge gained during the preparation stage will be documented in a LaTeX file that will be further used in the Preparation chapter of the dissertation.

D.3.2 Implementation of the multilayer perceptron neural network architecture

Deep learning methods generally require large datasets because the neural networks have a complex structure with a lot of weight parameters that need to be optimised. Figure 1 illustrates the architecture of a feedforward neural network with two hidden layers. For the epigenetic data, the input layer will consist of the gene expressions in the dataset while the output layer will have the results of the particular inference task performed.

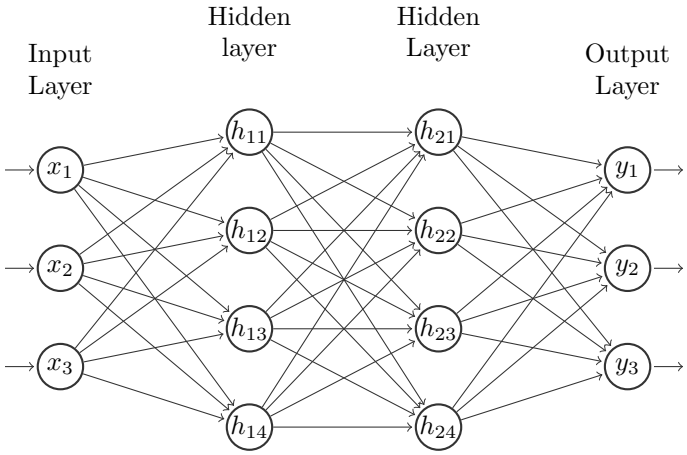


Figure D.1: Feedforward Neural Network

One of the main issues when using deep learning on sparse datasets is the high risk of overfitting the training data. Overfitting results in the neural network learning the particularities of the training data but being unable to generalise on test data.

Regularisation techniques can be used to obtain good generalisation performance in sparse datasets:

- *Weight decay*: Adds a penalty term to the loss function to ensure that the weight parameters converge faster to smaller absolute values.

- *Dropout*: Reduces the complexity of the neural network by randomly removing hidden units during training.

Moreover, the neural network architecture contains many hyperparameters such as the number of neurons in the hidden layers, the number of hidden layers, the way in which the neurons in the hidden layers are connected to each other as well as the the activation functions for the neurons.

Regularization and optimisation of hyperparameters to improve the performance of the network on the inference task will be the primary focus of this stage.

D.3.3 Implementation of the recurrent neural network architecture

Epigenetic data can be modified to be processed sequentially. This will require a preprocessing step that will simply involve reordering the genes from the data in a suitable manner for this sequential training.

Recurrent neural networks can be used to take advantage of this sequential structure of the epigenetic data. [18] As opposed to feedforward network mentioned earlier where the data flows in a single direction, recurrent neural networks have loops as illustrated by Figure 2.

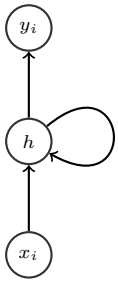


Figure D.2: Recurrent Neural Network

These loops allow the information to persist between sequent states. Figure 3 illustrates the recurrent neural network unrolled and how the state of the hidden layer is transferred between the consecutive steps of the computation. The input to the network will consist of sequences of gene expressions.

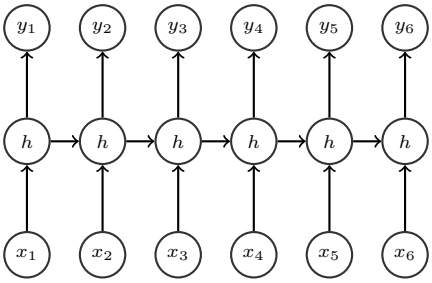


Figure D.3: Unrolled Recurrent Neural Network

This special architecture of the recurrent neural networks that permits information to be transferred between states requires fewer parameters that need to be learned, because the parameters are shared, which is particularly useful for sparse datasets.

Therefore, the tasks performed will be preprocessing the data so that it can be analysed in sequential order as well as implementing the layers and the connections between the layers for the recurrent neural network architecture.

D.3.4 Implementation of the superlayered neural networks architecture

The structure of epigenetic data will be exploited even further in this stage. Using unsupervised learning, clustering of the genes will be performed for additional feature extraction. Clustering can be used to discover the underlying structure of the data. For instance, the genes can be clustered based on their expression levels in the cells in order to find groups of co-expressed genes.

Subsequently, a superlayered neural network architecture can be developed using these clusters.

The superlayered architecture works as follows: each cluster of genes is used as input features to train one feedforward neural network in the architecture. [33] Then, the neurons in each superlayer are cross-connected to neurons in the other superlayers as illustrated in Figure 4.

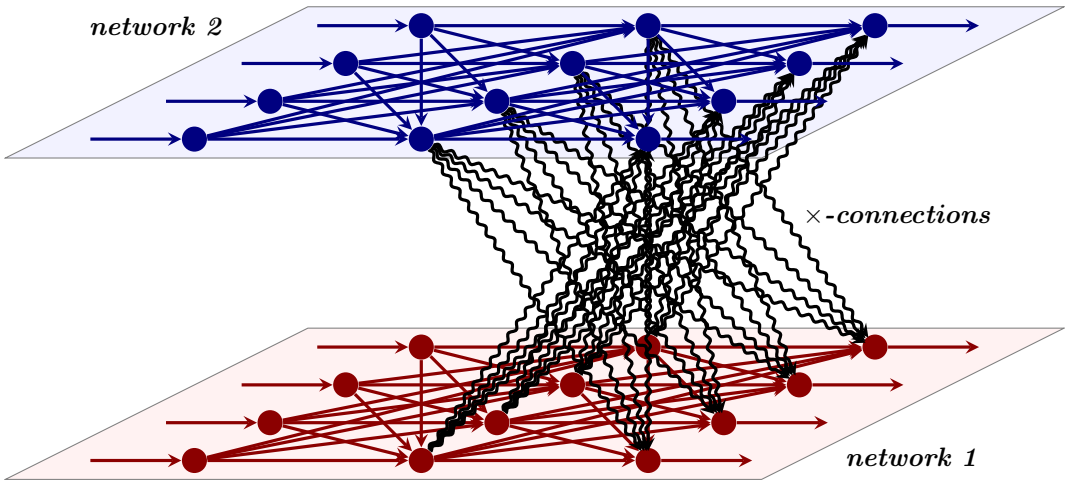


Figure D.4: Superlayered Neural Networks

The cross-connections between the layers of this architecture will explore the relations between the different gene clusters.

The specific subtasks that will be performed at this stage involve implementing a clustering algorithm for the genes, determining the number of superlayers, adjusting the hyperparameters of the feedforward neural network architecture previously implemented so that it can be used as a superlayer and implementing the cross-connections between the superlayers.

D.3.5 Evaluation of the architectures

The epigenetics dataset is in matrix form with the genes representing the rows and the biological cells representing the columns. A value in the matrix represents the expression level of the gene for the particular cell. Each column has a label which can be used for supervised learning.

The dataset can be used in supervised learning for predicting the label for each sample based on the gene expressions of the cell. Therefore, the dataset will be divided into a training set and test set for evaluation purposes.

The same dataset will be used for all the architectures in order to be able to produce a valid comparison between them.

Assessment metrics such as the F-measure, the area under the receiver operating characteristic curve (AUC) and the area under the precision recall curve (AUC-PR) will be used for the evaluation and comparison of the models.

D.3.6 Dissertation write-up

The dissertation write-up will also happen concurrently with the other stages of the project. The purpose of this stage is to document the work done by writing a dissertation that will contain the following chapters: Introduction, Preparation, Implementation, Evaluation and Conclusions.

D.3.7 Possible extensions

- ***Test different clustering methods for the superlayer approach*:** each clustering method analyses the structure of the data in different ways; thus it would be beneficial to notice how different clustering algorithms affect the overall results of the inference task performed on the data.
- ***Use the gene clusters as input sequences to the recurrent neural networks:*** clustering exploits the underlying structure of the data, while the recurrent neural network takes advantage of its sequential property; using the clusters as input sequences to the recurrent neural networks may give valuable results.

D.4 Success criteria

The success of the main project will be evaluated against the following criteria:

- Implement the neural network architectures described above.
- Test the correctness of the models using synthetic data.
- Evaluate each architecture on epigenetic data and compare the way in which each architecture exploits various features of the data.
- Utilise good assessment metrics to clearly evaluate how limited and imbalanced data influences the performance of the deep learning architectures used.
- Write the dissertation to present the work done and the corresponding results obtained.

In addition, the extensions of the project should also be achievable. The success criteria for the project extensions are:

- Implement at least another clustering algorithm that can be used in the superlayered architecture and evaluated on the data.
- Evaluate the recurrent neural network architecture when the input sequences are represented by the gene clusters.

D.5 Resources required

I will use my own laptop (2.6 GHz Intel Core i5 with 8GB RAM running Ubuntu) to write the source code for the neural network architectures, to evaluate them and to write the final dissertation.

The following contingency plans will be used:

- Revision control for the code and the dissertation using git by regularly pushing the code into a repository on Github.

- Regularly uploading all the documents to Google Drive and Dropbox.
- Regularly saving all the files necessary for the project on an external hard disk.

Epigenetic datasets are available online:

- The Cancer Genome Atlas: http://cancergenome.nih.gov/
- Gene Expression Omnibus: https://www.ncbi.nlm.nih.gov/geo/

Moreover, Dr Pietro Lio' will provide me access to GPUs in case they will be needed for faster training of the neural networks.

D.6 Timetable

The project will be split in slots as illustrated below. The planned starting date of the project is the 21st October 2016, after the submission of the project proposal. Each slot has several milestones that illustrate the successful completion of the work allocated for the respective slot.

The timetable is designed such that the project is implemented and the dissertation is ready for submission by the end of the Easter vacation, so that I can use the remaining time revising for the written examination. In addition, the timetable allocates slots for the implementation of the project extensions. In case of a schedule overrun, the time from the beginning of Easter term until the dissertation deadline on the 19th of May can be use as slack time.

- **Michaelmas weeks 3–4 (21st of October to 2nd of November):**
  - Read books and online materials about neural networks to understand how the network learns to perform the inference tasks.
  - Read papers about using deep learning for bioinformatics to learn the best practices for choosing the network hyperparameters for epigenetic data.
  - Become acquainted with the Python API of TensorFlow.

  Milestones: Have a good understanding for implementing the multilayer perceptron neural network, document readings so that I can use the information in the Preparation chapter of my dissertation and consult with my supervisor to ensure that my understanding is correct.

- **Michaelmas weeks 5–6 (3rd of November to 16th of November):**
  - Implement the feedforward neural network architecture.
  - Optimise the architecture to account for data sparsity.
  - Test it on the synthetic data and on the epigenetic data.

Milestones: Document the way the architecture learns from data and have a working model that achieves the required accuracy on the synthetic data and that be evaluated on the epigenetic data.

- **Michaelmas weeks 7–8 (17th of November to 31st of November):**

  * Read papers and books to learn how the architecture of the recurrent neural networks takes advantage of the sequential format of the data and how the network learns from this type of data.
  * Prepare the synthetic and epigenetic data to be processed in sequential order.

  Milestones: Document the recurrent neural network architecture, have the data readily available for testing when the model is ready and consult with supervisor to ensure my understanding of the recurrent neural networks is correct.

- **Michaelmas vacation weeks 1–2 (1st of December - 14th of December):**

  * Implement the recurrent neural network architecture.
  * Test the model on the synthetic data and on the epigenetic data.

  Milestones: Have a working recurrent network that achieves the required accuracy on synthetic data and that can be evaluated on the epigenetic data.

- **Michaelmas vacation weeks 3–4 (15th of December - 21st of December):**

  * Buffer slot: complete any remaining implementation.
  * Create a LaTeX document with the layout of the dissertation.
  * Start writing up the Introduction, Preparation and Implementation chapters.

  Milestones: Have a LaTeX document for the dissertation set up.

- **Michaelmas vacation weeks 5–6 (22nd of December - 4th of January):**

  * Read papers about clustering algorithms used for epigenetic and biomedical data.
  * Read papers about superlayered neural networks architectures and the way they have been previously used.
  * Start implementing the chosen algorithm for clustering the genes.

  Milestones: Write a document on the cluster algorithm chosen along with the appropriate justifications for it, include details of the superlayered neural networks architecture and consult with the supervisor to ensure correct understanding.

- **Michaelmas vacation weeks 7–8 (5^{th} of January - 18^{th} of January):**
  - Finish implementing the chosen algorithm for clustering the genes.
  - Start implementing the superlayered network architecture.
  - Test the clustering algorithm on both the artificial data and the epigenetic data.
  - Start writing progress report.

  Milestones: Have a working clustering algorithm that achieves the required accuracy on the artificial data and that can be used on the epigenetic data.

- **Lent weeks 1–2 (19^{th} of January - 1^{st} of February):**
  - Finish writing progress report.
  - Prepare presentation slides.
  - Finish implementing the superlayered architecture.
  - Test it on artificial data and epigenetic data.

  Milestones: Submit progress report and have a working superlayered architecture that achieves the required accuracy on the synthetic data and that can be used on the epigenetic data.

- **Lent weeks 3–4 (2^{nd} of February - 15^{th} of February):**
  - Rehearse and deliver project presentation.
  - Set the framework for evaluating the three proposed architectures and compare their performance.
  - Use several assessment metrics (such as F-measure, AUC, AUC-PR) to evaluate how each architecture performs on the sparse and imbalanced data.
  - Write up the results in the Evaluation chapter of the dissertation.

  Milestones: Successfully deliver project presentation and successfully evaluate the models.

- **Lent weeks 5–6 (16^{th} of February - 1^{st} of March):**
  - Work on the first extensions of the project—Implementation of additional clustering algorithms.
  - Test the implemented clustering algorithm on the synthetic and epigenetic data.
  - Evaluate the superlayerd architecture using the new clustering of the genes.
  - Incorporate all the documentation in the Preparation chapter of the dissertation.

Milestones: Have at least another clustering algorithm implemented and successfully evaluate the superlayered architecture using the new clustering algorithm.

- **Lent weeks 7–8 (2^{nd} of March - 15^{th} of March):**
  - Buffer slot: finish any remaining implementation.
  - Work on the second extension of the project—Use the clusters as input sequences to the recurrent neural networks and evaluate the results.
  - Write in the Implementation and Evaluation chapters of the dissertation.

  Milestones: Successfully evaluate the recurrent neural network architecture when the input sequences are represented by clusters.
- **Easter vacation weeks 1–2 (16^{th} of Match - 29^{th} of March):**
  - Complete the first draft of the dissertation.
  - Submit it to supervisor and Director of Studies for feedback.

  Milestones: Successfully submit the first draft of the dissertation.
- **Easter vacation weeks 3–4 (30^{th} of March - 5^{th} of April):**
  - Proofread and make improvements to the dissertation.
  - Take comments on the first dissertation draft into consideration and amend accordingly.
  - Complete second draft of dissertation and submit it for feedback.

  Milestones: Successfully submit the second draft of the dissertation.
- **Easter vacation weeks 5–7 (6^{th} of April - 18^{th} of April):**
  - Take comments on the second dissertation draft into consideration and amend accordingly.

  Milestones: Have dissertation ready for submission.
- **Deadline: 19^{th} of May**